

AD-A089 090

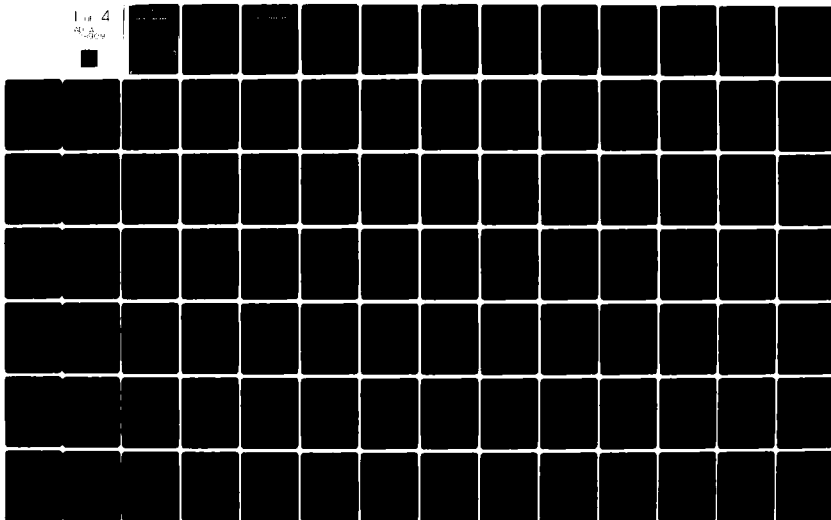
CALIFORNIA UNIV IRVINE DEPT OF INFORMATION AND COMP--ETC F/G 9/2
PROCEEDINGS OF THE IRVINE WORKSHOP ON ALTERNATIVES FOR THE ENVI--ETC(U)
1978 T A STANDISH
DAA629-78-M-0219
UCI-ICS-78-83

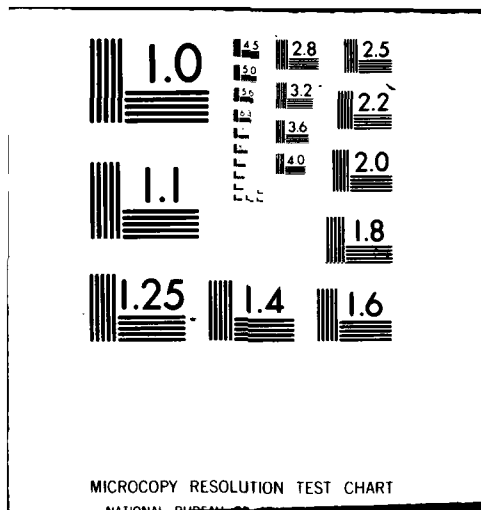
UNCLASSIFIED

NL

1 of 4

AD-A089 090





LEVEL II

(C)

**Proceedings of the
IRVINE WORKSHOP**

**on Alternatives for
the Environment, Certification, and Control
of the**

DOD Common High Order Language

June 20-22, 1978

University of California at Irvine

AD A089090

**DTIC
ELECTE
SEP 10 1980
S D A**

**CO-SPONSORED BY THE
U.S. ARMY, U.S. NAVY, U.S. AIR FORCE
AND THE
IRVINE COMPUTER SCIENCE DEPARTMENT**

DISTRIBUTION STATEMENT A

**Approved for public release;
Distribution Unlimited**

80 6 5 011

DDC FILE COPY

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(12) 335

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER CS-78-143	2. GOVT ACCESSION NO. AD-A089090	3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE (and Subtitle) PROCEEDINGS OF THE IRVINE WORKSHOP ON ALTERNATIVES FOR THE ENVIRONMENT, CERTIFICATION, AND CONTROL OF THE DOD COMMON HIGH ORDER LANGUAGE Held at California University, Irvine on 20-22 June 1978		5. TYPE OF REPORT, PERIOD COVERED final report, 1 JUN-78 31 DEC 78
6. AUTHOR(s) (10) Thomas A. Standish (editor)		7. PERFORMING ORGANIZATION NUMBER (15) DAAG29-78-M-0219
8. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of California at Irvine Irvine, California 92717		9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (11)
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. ARMY RESEARCH OFFICE, PO BOX 12211 RESEARCH TRIANGLE PARK NORTH CAROLINA 27709		12. REPORT DATE JUNE 20-22 1978
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (14) UCI-ICS-78-83		13. NUMBER OF PAGES 358
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		16. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) This document is approved for public release and sale; distribution is unlimited. This document may be reproduced for any purpose of the United States Government.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) -----		
18. SUPPLEMENTARY NOTES -----		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Ada Environment, Programming Language Standardization, Programming Language Specification, Program Verification Technology, Compiler Validation Technology, Compile Time Tools, Requirements Analysis, System Design, Program Documentation, Program Maintenance, Software Test and Measurement, Program Development Systems		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The Proceedings contain edited transcripts and position papers presented at a Workshop on Alternatives for the Environment, Certification, and Control of the DoD Common High Order Language (Ada), held June 20-22, 1978 at the Irvine campus of the University of California. Topics discussed include: experience in language standardization, technology for language specification, verification technology, compile time tools, supporting a programming language culture, requirements analysis, system design, program documentation, maintenance, program development systems, and test & measurement.		

Proceedings of the
IRVINE WORKSHOP

on Alternatives for
the Environment, Certification, and Control
of the

DOD Common High Order Language

June 20-22, 1978

University of California at Irvine

DAG 29-78-M-0219
(ARO)

CO-SPONSORED BY THE
U.S. ARMY, U.S. NAVY, U.S. AIR FORCE
AND THE
IRVINE COMPUTER SCIENCE DEPARTMENT

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DOD TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	<input type="checkbox"/>
<i>Letter on file</i>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
<i>A</i>	

Source of Support and Notices

The Workshop and the publication of the Workshop Proceedings were supported by the United States Army under contract DAAG29-78-M-0219.

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Note to Reader Added in Proof

Throughout the transcripts, the Workshop participants used the name "DOD1" to refer informally to the DoD Common High Order Language. As noted by Col. Whitaker in his Opening Session Address, "DOD1" was not the name of the language at the time the Workshop was held, nor has it ever been the name of the language. Since the end of the Workshop, the name Ada has been chosen as the name for the common language. This honors Ada Augusta, the Countess of Lovelace, the daughter of the poet Lord Byron, and Babbage's "programmer." For reasons of historical accuracy, the use of the working title "DOD1" has been retained in the transcripts. The reader should be aware, however, that the participants were informed by Col. Whitaker and were aware that "DOD1" was not the official name, and that they used "DOD1" as a term of convenience only in the absence of any other suitable working designation.

COPYRIGHT © 1978, THOMAS A. STANDISH, ALL RIGHTS RESERVED

EXCEPT AS FOLLOWS:

This document may be reproduced for any purpose of the United States Government.

Table of Contents

	page
Acknowledgments	i
Introduction	1
Opening Session	4
Session 1A: Experience in Language Standardization	15
Session 2A: Technology for Language Specification	25
Session 3A: Verification Technology, Present & Future	38
Session 4A: Technology for Compiler Validation	52
Session 5A: Compile Time Tools	59
Session 6A: Supporting a Flourishing Language Culture	73
Session 1B: Requirements Analysis	90
Session 2B: System Design	100
Session 3B: Program Documentation	116
Session 4B: Program Development Systems	125
Session 5B: Program Maintenance	139
Session 6B: Test and Measurement	149
Session 1C: Training and Education	158

Position Papers

Position Paper --- Richard N. Taylor

Position Paper --- John Burgey, John Machado,
John Perry, and Patricia Santoni

Levels of Program Debugging --- Robert Balzer

Maintenance --- Robert Balzer

Toward Self-Documenting Programs --- Edward A. Taft

Acknowledgments

Many fine people deserve credit for contributing to the success of the Workshop and to the preparation of the Proceedings.

Most deserving of praise are Mary Kay Clarke, Secretary to the Chair, and Phyllis Siegel, Administrative Assistant, both of the Irvine Computer Science Department. Mary Kay and Phyllis arranged the Workshop facilities, planned the meals and refreshments, handled mailings and travel arrangements for participants, handled finances, and spent the summer typing transcripts. Their skill and devotion are the principal factors that contributed to the success of the entire undertaking.

Katie Heap and Randi Steinman of the UC Irvine Conference Office were most helpful in arranging on-campus lodging, providing audio-visual aids, and reserving Workshop conference rooms. Alexander Hu, Dennis Kibler, Craig Taylor, and David Smith helped run the audio-visual equipment during the working sessions and were responsible for taping the conversations and presentations.

A number of good people in and surrounding the military services were marvelously supportive and helpful in arranging financial support for the Workshop and in planning the Workshop Agenda. These include Warren Loper, Sam DiNitto, Jim Wagner, Russ Eyres, Dave Fisher, Bill Whitaker, and Jimmy Suttle.

With regard to the Proceedings, we should have known we were in for trouble when we were unsuccessful in hiring courtroom stenographers to produce the transcripts of the Workshop sessions. That the stenographers wanted \$75 per hour was no problem --- we had long since resigned ourselves to the notion that anybody connected with the legal profession would charge totally outrageous fees and we were prepared to pay through the nose in the fine company of others of our suffering countrymen. The problem we could not surmount was that the stenographers flatly refused to transcribe technical material (though how it is that legal jargon is less technical than that of computer science completely escapes us). Faced with their refusal, we asked "Did they know something we didn't know?"

We had foreseen that there would be problems with non-technical people trying to transcribe technical jargon, and we were prepared to see "Wirth's Euler" come out as "worth's oiler" and to see "BNF" come out as "B&F" in the draft transcripts. These minor irritants we thought we could expunge with a few magic TECO macros once the transcripts were captured in the computer. Alas, the computer went down in mid-summer to get an enlarged memory, and we badly underestimated the magnitude and difficulty of the task. Furthermore, our recording technique was less than flawless.

It took all summer to get the transcripts typed. Mary Kay Clarke, Tammy Feldman, Shirley Rasmussen, and Alexander Hu stuck with it to the bitter end. Tammy Feldman's extraordinary intelligence and devotion led to very high quality draft transcripts and her quiet competence deserves the warmest praise. We repeatedly hired outside help to speed up the process, but they repeatedly quit after a week of our special brand of secretarial torture.

By Sept. 30, we had a four inch stack of draft transcripts completed, and we mailed out a four foot stack of them to the Workshop participants for editing. We got back three feet of edited transcripts by the end of October.

Now it was time for the suffering to penetrate other layers of our Department as we attempted to boil down each session into eight to twelve pages of the best material. This called for help from technically qualified people who could judge relevance and interest, who could compress detail, and who could turn informal, run-on conversation into legible English. While some semantic distortion is inevitable in this process, the crew that we had working on it did a splendid job, and we believe we have minimized the semantic distortion that could have resulted had less capable people been involved. In this phase, Jim Meehan of the Irvine Computer Science faculty, and Irvine graduate students Paul Mockapetris, Ashok Viswanathan, David Smith, Essam Hosny, and Gene Fisher were each indispensable. Gene Fisher deserves special praise for his superlative contributions at this stage.



Thomas A. Standish
Workshop Coordinator

Introduction to the Workshop Proceedings
by
Thomas A. Standish
Workshop Coordinator

It is tempting to begin this Introduction with a Lincolnesque prediction that history will little note nor long remember what we did at this Workshop.

The Workshop was hastily organized under pressure to provide timely input and advice to a preliminary stage of the PEBBLEMAN environment requirements process. There was insufficient time for the participants to prepare careful advance technical papers for presentation. So, for the most part, the participants came equipped only with their own experiences, and with attitudes of cheerful curiosity, a desire to help, and a zest for exploration of new issues.

Everyone seemed aware that we were getting our toes wet together in a new and profound bunch of issues that needed careful thinking and that needed the best in wisdom we were able to provide. In retrospect, the fact that so many of the participants did not prepare technical papers for delivery, and so were not in an ego-involved mode of pushing their own intellectual wares, led to a spirit of cooperation, open-mindedness, and eagerness for learning that seemed, at times, quite magical --- especially given the tri-lateral, military-industrial-academic origins of the participants. In fact, as the reader will note, the spirit of the Proceedings is remarkable for the conspicuous absence of traditional chips on traditional shoulders.

What emerged, then, was an astonishingly broad range of coverage of interesting issues with interesting observations, data, experience, and ideas coming from many origins. The Proceedings capture for the reader a good deal of the richness of what was discussed, and they are remarkable for their scope of coverage. In my judgment, the Proceedings contain a tremendous amount of fertile thought, and constitute essential reading for anyone attempting to acquire the "background context" or to "get spun up" in environment issues. Most importantly, perhaps, the Proceedings will help to sensitize the reader to the staggering breadth of issues that need to be considered in the environment requirements process.

So perhaps the Lincolnesque view of the Workshop is a bit too modest. While they may not be earthshaking, the Proceedings capture for the benefit of those not present, one of the first attempts of a tri-lateral group of participants to come to grips with environment issues, and they record some of the spirit of the social interactions as well as the ideas and experience generated by this unique group addressing this unique set of issues for the first time. It does not take too much crystal ball gazing to see that environment issues will become increasingly prominent in the 1980s. Thus, if you were not there, the Proceedings

capture for you some first attempts to embark on development of the Environment as a coherent, new subfield of investigation in Computer Science.

The Workshop was organized around two series of six parallel sessions. These parallel series consisted of the "A" Sessions (1A, 2A, ..., 6A) dealing with language standardization and specification, compiler validation and verification, and support of language cultures, and the "B" Sessions (1B, 2B, ..., 6B) dealing with software life cycle issues (requirements analysis, design, documentation, program development systems, maintenance, testing, and measurement). A few interested participants formed a third parallel series, the "C" Sessions, one of which dealt with training and education and one of which dealt with the social settings in which use of the Common Language is likely to take place. In addition to the parallel sessions, Warren Teitelman of XEROX PARC presented a color-TV film illustrating use of the INTERLISP programming environment before a joint session of the entire Workshop.

The conversations that took place during all of the working sessions (except that on social settings) were recorded and transcribed, and the participants were invited to return edited transcripts for consideration for inclusion in the Workshop Proceedings. Of necessity, these edited transcripts had to be compressed quite a bit to form a product of manageable size. It is inevitable that some semantic distortion has crept into these summarized, edited remarks, due to technical errors in transcription, loss of material through summarization, and the semantic maneuvering needed to turn spoken dialogue into legible English. Nonetheless, the flavor of what was said seems to have been captured moderately faithfully even if the individual wording has been markedly changed from the original, in some cases.

In an ideal world, we would have liked to submit the current transcripts to the participants for a final round of editing, but time has prevented us from taking this last step. Thus, we apologize, in advance, to those whose remarks got twisted beyond recognition or taken totally out of context, and to those into whose mouths we placed words they did not speak in order to smooth the flow of the conversation and render the written transcripts legible. We hope those so victimized will trust that we had no malice in our hearts as we struggled to produce a legible product.

The participants were invited to bring "position papers" to the Workshop and to submit them for publication in the Proceedings. A number of these have been reproduced in the second part of this volume. In some cases, the position papers were written at night during the Workshop. In one case, position papers covering the ideas generated in the Parallel "C" session on social settings was contributed after the Workshop, since that session was not recorded and transcribed. These papers serve to replace the edited transcripts that would have been generated had that session been recorded.

The participants were also urged to comment on the draft PEBBLEMAN document circulated prior to the Workshop, and to submit proposed changes to the preliminary document for possible inclusion. A number of such candidate changes were submitted as a result of the deliberations of the Workshop, and the Chairmen of each of the Working Sessions were invited to read aloud the relevant sections of the Preliminary PEBBLEMAN document at the outset of each session, and to solicit the views of the session participants.

The Opening Session Address by Lt. Col. William A. Whitaker is particularly important for understanding the background and setting of the Workshop, and readers are urged to study this address first in order to gain a proper initial perspective.

Opening Session
 Welcoming Remarks by T. Standish
 Opening Session Address by Lt. Col. William A. Whitaker, USAF

Welcoming Remarks
 by
 T. Standish

On behalf of the Computer Science Department, I would like to welcome you to the Irvine Campus of the University of California.

This is the campus where some chemists made the discovery that fluorocarbons wreck the ozone layer. While some say this discovery has saved the entire future of the planet, others feel that the major impact has been to pelt you with a lot of new deodorant ads, such as the one which urges you "To get off your can and on the stick".

A word about our sponsors --- This workshop is sponsored jointly by the Army, Navy, and Air Force, and, as such, is a venture of the joint Services

I would now like to introduce our speaker for the opening session, who is Lt. Col. William A. Whitaker.

In connection with the Common High Order Language, Col. Whitaker works at the pleasure of the Undersecretary of Defense in the capacity of Chairman of the High Order Language Working Group (HOLWG). He has been involved with computers since 1953, when he cut his programming teeth on a 604. He has a Ph.D. in Physics from the University of Chicago and he has probably consumed more computer time than anyone in this room. At one time he had one and a third 6600s at his disposal full time for a period of six to seven years, and he has accumulated personally ten man years on a 6600. If any of you can match that you're welcome to try.

Opening Session Address
 by
 Lt. Col. William A. Whitaker

Thank you. I will quickly bring people up to date, giving a very fast overview of the program. I would like to give the entire one hour talk in about 10 minutes, so, if the view graphs move too fast for you, come up and get them later, because I'm not going to keep them on very long. We're talking about the DOD Common High Order Language effort, and we're talking about what we call embedded computer systems. This has a strange legalistic definition, but generally they are weapons systems, communication systems, command and control, avionics, simulators, real time systems, and systems that are rigidly attached to some very large overall system. We are not talking about what we

at the DOD call automatic data processing equipment --- that is, financial management, inventory, accounting, payroll that sort of thing. In particular, that's done by COBOL in the DOD just like it is everywhere else. We are not talking about large scientific computing which is done by FORTRAN, just like it is everywhere else. I emphasize this all the time because it is in our charter that we're not in the Common Language effort to replace COBOL or FORTRAN. In fact, it is the success of COBOL and FORTRAN that encourages us in this work. We are not trying to compete.

I will also remind you that we're involved in programming languages --- programmers talking to computers, not special application packages or simulation languages like GPSS or SIMSCRIPT, not automatic test equipment languages like ATLAS which is a test designer to technician language. We're talking specifically to computers, not talking to data bases or whatever. The High Order Language Working Group is running this for the DOD. The members are the Office of the Secretary of Defense and the Services. It was specifically charged to formulate requirements for DOD high order languages, evaluate the existing languages, and recommend implementation and control of a minimal set of high order languages. One recommendation is a nice minimal set, but that's not the way it started out.

When we started out, we had no control, no commonality at all in the DOD. We did not even have high order languages, for all practical purposes. A very small amount of what was done in the DOD was in high order languages. So we started out and made a DOD Directive and said you're going to use high order languages unless we know the reason why; and it's going to be hard for you to find a reason why because the majority of the money we spend is in maintenance, not in program development, not in writing the programs, but in maintaining them for 20 years. High order languages obviously have great advantages in that area over assembly languages. It further says that the approved DOD high order languages will be assigned to a control agent. We're not going just to name them on a piece of paper, we're going to control them. And the interim list of approved high order languages was given as: COBOL, FORTRAN, CMS-2, SPL1, TACPOL and JOVIAL-J3 and J73. These were the languages chosen from the hundreds of languages we had beforehand that we restricted things to. That certainly is a small number compared to what we had, perhaps not a minimal number, but a small number. Why aren't we happy with that?

Languages on that list are not necessarily the highest technology, the most appropriate, or the most powerful languages you'd like. They are not satisfactory. Beyond this administrative solution, we need technical advances. We need high order languages that reduce the total life cycle cost of software, that promote responsiveness, timeliness, flexibility, reliability, etc. Reliability is obviously very important for the DOD. Maintainability and efficiency are also important. Commonly, the situation has been in the past that high order languages have been

rejected because they produce inefficient code and everything has to be squeezed down into that tank, that plane, or what have you. The old argument says that next year core will be twice as cheap and computers will be twice as fast. But this doesn't make much difference if you have already bought your computer, or if, in fact, you bought 6000 of them, and you've got to squeeze everything into those little boxes. You don't get to buy a new one next year.

Well, those are nice wish list kinds of things but they're not very quantitative. We have gone through an exercise to define the requirements on a functional level, that is, not so specific that they really define a special language, not so general that you can't define them. We have done this by an iterative procedure. We generated a document called STRAWMAN, and that was circulated, commented on, and improved and called WOODENMAN. The next one was TINMAN. TINMAN had an interesting property. It was noted at that point that all the applications areas that we were dealing with could be satisfied by a single set of requirements. We started off thinking there was going to be a set of requirements for avionics, a set of requirements for simulators, for climatic control, etc. When we got down to that stage, there was only one set of requirements, both necessary and sufficient, for all the applications. We finally understood that, but it was a non-trivial thing to start with. It doesn't say that there can be a language that satisfies all those requirements, just that if there were such a language, it would be the right one. We now have an IRONMAN (IRONMAN by the way is not the name of a language. We don't have a name for the language. DOD-1 is not the name of the language). IRONMAN has been revised, and next week we will have a STEELMAN. These are requirements for the language. The STEELMAN looks pretty much like a language manual. It has syntax, types, expressions, constants, control structures, procedures, and that sort of thing, but it is not a description of a language. It is a description of functional requirements that the language must meet.

We have gone through three economic analyses of the benefits of using the language, the introduction rate, the adoption rate, and that sort of thing. We're not here today to debate the advisability of doing this whole program. We've been operating on this program for three and a half years. It has been widely commented on and we have a number of independent economic analyses. We will have even more I am sure before we are through. All of the economic analyses agree that with an appropriate language and tools the DOD can save hundreds of millions to perhaps several thousands of millions of dollars. That seems like a fair amount of money, but then we're spending a fair amount of money, perhaps three or four billion dollars a year, on software as it is. So we can save an interesting fraction of that. There are technical benefits and there are commonality benefits. Commonality is often considered cost avoidance. That is, if you have a compiler that two people use, you didn't have to write it a second time.

The goals of the program are: A modern high order language in order that you can really do the program in high order language and not have to drop down into assembly language as we do now. Programming tools --- a language by itself is only a step; you really have to have the programming tools and to develop the environment in order to increase productivity. A common high order language allows you to share the expenses over a number of different programs and therefore reduces the cost. A minimal number reduces the cost to the smallest number. A single language however has unique advantages. You can expect a fair amount more cooperation from outside DOD if we could get down to a single language.

We evaluated the existing languages against our set of requirements, which was at that time contained in TINMAN. A number of languages were formally evaluated, and twice as many were informally evaluated --- most of the ones you can think of: COBOL, FORTRAN, TACPOL, CMS-2, CS4, JOVIAL J3 and J73, LTR, LIS, PASCAL, ALGOL 68, PL/I, ..., all sorts of languages. The group did not see any that we wanted to make the common DOD language, even with significant modifications. However, the evaluation group, the contractors, and the High Order Language Working Group were unanimous in the statement that it was desirable and feasible within the current state of the art to produce a single language meeting essentially all the requirements. This is the other side of the requirements statement. Not only if we had a language that met the requirements would that do it for DOD, but it now says that, on the basis of paper studies, we believe that the single language can be produced. So we went to a design phase to produce such a language.

Design contracts were let for the first phase of three phases. The first phase was a preliminary design. The second phase is a full design including a translator. The final phase is for initial maintenance and control, so we can get squared away. There were four Phase I contracts awarded for competitive prototyping. The contractors were CII-Honeywell Bull, Intermetrics, SofTech, and SRI International. You will note that all the successful bidders, and this was an open RFP on which we had a number of bidders, chose to start from PASCAL as a base. This was not a constraint on the contract, and other bidders chose other bases, but all four Phase I winners chose to start with PASCAL. This is a convenience for us. The Common Language is by no means to be a PASCAL superset. What we're designing against is the set of requirements that are very much different from the specifications for the design of PASCAL. PASCAL is supposedly a small language --- a teaching language. We obviously have other requirements. Nevertheless, there will obviously be a resemblance to PASCAL, but it is in no sense a PASCAL superset.

You will note that there are colors attached to the Phase I language designs --- green (CII-HB), red (Intermetrics), blue (SofTech), and yellow (SRI). For those, you have seen the product of the first phase, and

those were the contractors associated with them. At the end of the first phase we got a product --- the preliminary language designs. These were reviewed by 80 industrial, academic, and military teams. On the basis of these analyses, a decision was made to continue in a prototype fashion, two of the contractors --- CII-HB and Intermetrics. Now we are in the next stage of the exercise.

Having the design well under way (that is, we're pretty far down the line now), we can turn our attention to the development environment. We will probably spend 80% of the program resources on the development environment, and 20% on the actual language itself, so we are talking about a fairly extensive development outside just the language design. There are compilers. Obviously we need compilers in order to get acceptance of the language. One of the main difficulties with the languages we have now is that if you wanted to go out and use them in your program and you went out for bid on machines, and you've got this wonderful machine that is the low bidder, it's not going to have a compiler for your favorite language. In fact, it's probably not going to have a compiler for any language, and that's a difficulty. High order languages will only be popular in the DOD when compilers are widely available. We're going to write compilers, but more than that, we have to provide compiler writing technology for those compilers that we don't write.

We need tools of all sorts. In doing an audit of major software projects in DOD recently I have been very disappointed to find that there are very, very few tools in evidence, and those that are in evidence are little used. I have lived in programming environments in which the tools were very powerful and very useful, and an increase in productivity of orders of magnitude is possible for very specialized tools. Very few of these are available. Last Thursday, I was at an ACM/NBS Symposium on Tools for Improving Computing in the 80s at Gaithersburg. That was the title of the Symposium, and as far as I could determine, there were no papers on tools. However, they all recognized that was the reason they were there, looking for somebody else to tell them the way. Today, we are looking for you to tell us the way, and, in fact, to come up with tools. We need other supporting software, automatic translation aids, application packages, training, documentation, and all that sort of thing. PEBBLEMAN is the requirements document associated with the development environment.

Finally, I want to bring out the point that control of the language is extremely important. It certainly is something we have learned from COBOL and FORTRAN. You can't just publish an interesting paper and expect anybody to go out and write a compiler that looks anything like what you thought it was going to be. In the past, there has been a great deal of difficulty in defining the language so well so that the compiler writer could know, and there's been very little incentive to check on his being able to do so. In particular, the design of this language is not to be left to the compiler writer, which has been the situation in the

past. We will have validation of compilers. The systems that exist today, that work and that have improved the state of the art considerably are COBOL and CORAL 66 primarily. We will go through the formal procedures for standardization of the language. You will note that standardization is something that has been a slow process in other languages. Standardization has been impeded by the following process: you publish an interesting language, lots of people go out and implement it, all differently, then the standardization process has to get all these different compilers together. We're going to start off in a firmer position so that we don't ever generate that mess that takes 10 years to resolve, hopefully. Accessibility of tools obviously is an important part of control.

Let me go through the schedule. In August '77 the preliminary design contracts were let, April '78 the Phase I selection was made, April '79 the final selection will be made between the two remaining contenders. There will be test and evaluation of the language, not of the compilers, but of the language itself, something I don't believe has been done before. In this thing we expect to write a fair amount of code in a number of different application areas to test the suitability of the language and to find difficulties. One of the problems in the past is that it takes a long time for a language to settle out because it is written for one particular project which works with it for a while; and another project may come along and pick it up two or three years later, and they find all sorts of changes they'd like to see made. It is not so much a function of time as a function of the number of application areas and the amount of code that's been written. We're going to see if we can compress that to a very short time. Compiler implementations will take place during this period. In 1980 we're going to have availability of the language. By availability we mean that the language should be better supported in the number of compilers and tools and specifications than any other language on the approved DOD 5000.31 list. Then we will add it to the approved list.

I might point out just a few of the contacts we have made in this area. The European Community has been closely involved with our efforts. They started an effort that was very similar to what we're doing. They had political difficulties, were coming along fairly slowly, and have essentially abandoned that effort in order to follow us. The governments of the United Kingdom, France, and the Federal Republic of Germany have also contributed quite considerably to this effort. The International Purdue Workshop, LTPL-E and LTPL-A have supported us. We have Peter Elzer sitting here who is the former Chairman of LTPL-E and he represents the German Government.

Let me quickly go through the exercise for today. Environment requirements are what we're doing. The requirements for the environment are obviously in some sense looser than the language requirements. The language requirements are rigorous and specific and they are, in fact, requirements on the language. In some sense, certain

environment requirements are more of the situation of what we want to do, of a wish list of what we intend to do with our program. In some cases, particularly in the situation of control, they are very firm requirements, but it's a much broader spectrum in this area. We do wish detailed environment requirements, just as we have the requirements for the language, and we wish them to be widely circulated and commented on. We've been very open in this entire exercise, and that has been extremely useful. We say for public comment, but obviously that also means for comment inside the government and from government contractors. The DOD is involved in 50% of the software that's being developed in this country, so when you say "open comment", at least half of the people out there work for us anyway. I might point out that is in marked distinction to the situation with hardware, where we used to have a voting majority, but have it no longer. In software, we are still dominant. The requirements are to provide for management and control of the language. Obviously, the management issues that are addressed in the environment document are not really the concern of this Workshop.

The control of the language, particularly in the technical sense of control (i.e. how you do it), is obviously very important; and again, so are the tools. I might very briefly go through some philosophy, putting the situation in context. We do not intend to force the language or the tools on any systems project office, that is, speaking for the moment, for the High Order Language Working Group and for the Undersecretary of Defense. We regard the effort as successful when it is adopted voluntarily by these program offices because it is clearly better, cheaper, more available, more reliable, etc. That is not to say that somewhere else in the government, somebody may not require it, but that is not the purpose of this program.

We do not intend to provide all possible tools, just a basic tool box. We will encourage the normal software marketplace to produce tools and market them in the normal fashion, whether they be produced directly for the government, produced independently, rented, or what have you. We will be in the position of generating a market for these tools in the form of a common user to which it will be convenient to market. It is very difficult for the software industry now to market tools for J73, for instance, since there are only a couple of users of J73. Further, we do not wish to write all possible compilers, but the acceptance of the language depends upon very wide availability of compilers. We shall provide some compilers as necessary for the ongoing programs, and shall encourage others to build and validate their compilers through our validation system, particularly the machine manufacturers. We will provide compiler writing tools and facilities as we develop them that will be widely available. We will obviously not provide all training for the language. This will be done in the normal way; contractors normally provide their own training, the government provides internal training, and so forth. I must note that, in general, there is very little

training, and there is a fair amount of evidence that training does markedly increase productivity. A fairly recent study by IBM indicated that experience in training in the language itself, just that, can increase program productivity by a factor of two. The common occurrence in the industry is that a programmer hired off the street to work on the \$50 million effort gets a full four hours worth of training in the language. That's the state of the art. I don't believe that is cost-effective. We will provide materials for the training as necessary. That also indicates we may wish to produce such training materials and such other materials in different human languages, because we are looking for a certain international flavor in this exercise.

The PEBBLEMAN does not yet exist. There is a Preliminary Common Language Environment document which we have sent out to you. It is preliminary; there are lots of modifications we would like to make. It is something that has been fairly difficult to achieve, because it has been difficult to get people to reply with definite line-by-line changes. We hope that this Workshop will produce such. In fact, the comments and input from this Workshop will be incorporated into this document which will then become the PEBBLEMAN. Our time scale for this effort is that next week it gets typed. It is to be published by the 30th day of June, so we need the comments this week and we need them hopefully in a form which is appropriate to add to this preliminary document. I remind you, of course, that what we're talking about here is primarily the DOD environment, the development and maintenance environment. That is different from some others, particularly the academic environment. We are talking about building large programs, very long lived programs. We are talking, in some cases, about working in a different physical environment. For instance, there's an awful lot of use of cards in the DOD programming environment. Lots of people here haven't seen a card in a long time, nevertheless, there's a lot of them still around. Which is not to say that is the thing we have to stay with forever, but rather to realize there is a situation which we're trying to address. Not all comments from the Workshop may necessarily be appropriate for the PEBBLEMAN. Perhaps some of them of the more general nature are appropriate for the program management plan, which we're revising too. The present version of the preliminary PEBBLEMAN document we have here is primarily for guidance from this Workshop. We are not bound by anything in the preliminary document. Nothing is sacred --- neither the exact sections, nor the particular statements contained in them. Given expansion in detail, with specifics, we can later determine what is required, and what perhaps is just desired.

I guess that gives you what we regard as the importance of this exercise. There will be a Workshop Proceedings produced that will be valuable for other people engaged in this exercise later on. On the other hand, certainly the most concrete publication I can offer is that it will directly affect the DOD requirements as expressed next week.

So we do have a real need for this information.

R. Balzer: Can you say anything about the time frame that this Workshop should address in terms of creating, specifying, or portraying this program development environment?

Col. Whitaker: We normally are saying the program is "for the 80s"; that is, the language will become available in the 1980s. We note that from historical observation, a generation of either tools or language seems to be about ten years, although there is no firm commitment to that duration.

K. Bowles: You've given a very delicate treatment of the question of management. I, for one, wish you success. At the same time, note that DOD is not noted for voluntary action and you've laid out a plan that sounds like it depends on voluntary action. My question is, is there a way that we in this voluntary Workshop can strengthen your hand so that maybe there's a greater possibility of success?

Col. Whitaker: I tried to phrase my statements on that very carefully. This Group, the High Order Language Working Group, has no charter to enforce the use of the language. That is not to say that the Undersecretary of Defense doesn't have that charter. There are a lot of difficulties. We have people now who are committing themselves to the use of the language. On the other hand, there are a number that are sitting back and waiting, as well they might. There's a reason for a number of people not to commit themselves. If the language is technically successful, we certainly will have a large popular following. Then I think things will go very well. So the most important part that I recognize right now is really the technical success of the language. The difficulties that arise would be with individuals in the organizations. I would not, at this point, solicit your writing your Congressman on the matter. We are coming along very well. I might remark that on the basis of what is happening elsewhere, in other countries, it certainly appears that this is an effort whose time has come. There is wide recognition of that and so things are happening.

-----: On the question of standardization, has anything been done on subsetting? For example, will subsetting be allowed in this language at that time?

Col. Whitaker: We have not finally resolved that question, but tentatively, we see no reason to subset the language. We regard subsetting as a very dangerous thing because it leads to dialects.

-----: What's to prevent a particular contractor from using only

a portion of the language?

Col. Whitaker: Using only a portion of the language is a good thing. That is not subsetting.

-----: That's a subset.

Col. Whitaker: No, that's not a subset. There may be an administrative restriction that will not allow programmers to use a particular portion of the language on a particular project --- that's fine, because the program will compile on anybody else's compiler. However, what we consider subsetting is when the compiler itself refuses to recognize some construct. The validation procedure will require that every validated compiler recognize every construct of the language.

A. Gargaro: Going back to PEBBLEMAN, once this June 30th document is published, where do you expect to get the majority of feedback on the document --- from the participants of this Workshop, or from the DOD and the industry, at large?

Col. Whitaker: Presumably the participants of this Workshop will participate, but mostly those that did not participate would be involved. That's a much larger group.

-----: Do you have in the back of your mind a schedule saying when you would reconvene another Workshop?

Col. Whitaker: Not necessarily a Workshop. We would reissue the document when there are an appropriate number of comments. I would think it would not be before January 1979.

-----: What is the mechanism for distributing the document?

Col. Whitaker: We have a large mailing list. We talk about it wherever we go. People write in. It will be formally distributed through the Services. Everyone here will, of course, get a copy.

S. Crocker: On the subject of embedded computers versus non-embedded computers, are you trying to encourage, to discourage, or have you given any thought to the matter of technical implementation of the language on the ordinary machines we see around?

Col. Whitaker: That's a good question. "Embedded" has a very

special meaning in the DOD. The meaning is those computers that are brought under the 5000 series of regulations. The non-embedded computers are those reported under the Brooks Bill. This is not a hardware distinction. We have embedded computers that are 360s and 6600s, so the distinction isn't really that clear. Further, in the DOD there is a large use of cross compilers which are getting to be much more the common environment; so development programs are very common on 6600s and 360s, cross-compiling down to whatever your favorite mini is. Therefore, the compilers will more likely be hosted on these sorts of machines. The applications of writing payrolls which may be on the same machine type are not what we're interested in --- that's the distinction.

Session 1A: Experience in Language Standardization
 Cdr. John D. Cooper, Chair

- J. Cooper: We have two extreme methods within the DOD for controlling languages--the Navy's and the Air Force's. In the Navy we are the controlling agent of CMS-2, which has been around the longest. We have two compilers, one for 16-bit machines and the other for 32-bit machines. About 250 copies are installed around the world.

Copies of these compilers are installed and maintained by the Navy. The way the configuration management is performed is that we have only one copy of the compiler's source code and it's locked in a vault at the compiler maintenance facility, and nobody has access to it. When we go to a user site to install a compiler, we only give them an object load module. Therefore, since nobody has the source code, nobody diddles with the language. We do all the maintenance of the language for them so we know at all times the status of each of those 250 compilers. As a result, we have the burden of maintaining and providing the documentation for the language, as well as accounting for all the bugs in it. Now I'll let Sam DiNitto tell you how the Air Force controls JOVIAL, which is more like the COBOL way of doing things.

- S. DiNitto: Colonel Whitaker talked earlier about DOD directives 5000.20 and 5000.30. What happened was that each of the three services were supposed to come up with their own set of regulations governing the use of languages for themselves. Now the Air Force came out with a regulation called AFR300-10 which limited the number of languages that the Air Force could use to a small number, namely FORTRAN, COBOL, and JOVIAL. It excluded CMS-2, SPL1, and others. The Air Force was designated to control JOVIAL within DOD. That means that any DOD user would come to the Air Force for information about JOVIAL. The Air Force in turn designated a systems command group, which we are a part of, to be the control agent.

Unlike the Navy JOVIAL is not frozen. I believe it is 5000.31 that allows you to make changes to the language not more than once per year. The Air Force fully accepts that, which is wrong in my opinion. We still have a mechanism, that Colonel Whitaker mentioned, by which you can avoid using JOVIAL if you have a good enough reason for not doing so. So far, since the control mechanisms were instituted, we see more reasons for not using JOVIAL than reasons for using it. We set up this elaborate mechanism which consists of a designated control agent, and he is assisted by a policy control board ... which regulates how the language is going to be controlled.

The designated control agent is strictly a bureaucrat. Technically he doesn't have much to say. So he has delegated certain tasks to a language control agent, in this case for JOVIAL. This is the organization I belong to. It is responsible for performing all the technical duties associated with controlling the language. These include

validation of compilers. One thing I should point out about validation is that we will not validate or certify a compiler forever. It will only be validated once per application. The principal reason for that is that changes may come into the language and we want the new user of the language to get the latest version.

Assisting the control agent is an organization which is going to be called the Language Control Facility. It will collect data on the use of the language and information of that sort which we hope will give us some insight into where the problems are in the language. It will provide programming tools for the language (whichever ones are available) and give assistance in buying compilers.

[Mr. DiNitto goes on to describe the language control bureaucracy in the Air Force. He concludes by observing that it will probably be difficult to force the various control agencies to strictly adhere to regulations regarding the use of a standard DOD language and its associated tools.]

- J. Cooper: I want to set the record straight on a couple of things. Our language is not at all frozen. It changes not infrequently, but the major point is that there is a Navywide Configuration Control Board that manages the changes. The proposed changes are submitted to the board. We approve, or disapprove them, and then they are implemented by the compiler maintenance group. So when we have a change to the language or to the compiler, only one organization makes the change and then everybody has the same change.
- P. Wegner: Do you have any quantitative measure as to how many changes are made or how large they are?
- J. Cooper: Well, we have one basic ground rule and it goes without saying that DODI will have to have the same ground rule, that all changes have to be upwards compatible. So you don't do anything to impact any existing systems.
- P. Wegner: Are there several changes a week, several a month?
- J. Cooper: No, maybe eight to fifteen a year. We've always controlled the language by controlling the source code, but we've only managed it on a Navywide basis over the last four years. Since we have done that, it has served to limit the number of changes. Before that, the compiler maintenance activities performed their own configuration management, and made whatever changes they wanted to. Now that it is controlled on the Navywide basis, it has served to reduce the number of changes.
- S. DiNitto: When a so-called upward compatible change is put in, we found often that all the software has to be recompiled. This is not a minuscule task.
- J. Cooper: That can be a problem, but we only have the two compilers, and when we go to evaluate a change we know,

really know, what that change will involve and whether it would cause some subtle incompatibility. If it is going to cause a subtle incompatibility we don't implement it.

M. Wolfe: How many users do you have?

J. Cooper: Well, we have over 250 compilers installed, so that means that there are approximately 250 geographical locations that are doing Navy software. On the average there are 40 or more programmers at each location. So we've got tens of thousands of users.

-----: These two compilers, do they only operate on two separate machines?

J. Cooper: I was afraid you were going to ask that! The Navy is very standardization oriented. We have a standard language; we have standard computers; we have documentation standards that everybody has to follow. So we have a limited set of computers we have to generate code for. The 32-bit variety is hosted on our standard computer, the AN/UYK-7, and it only generates code for the AN/UYK-7. However the one for 16-bit word size is quite different. It's written in standard FORTRAN, it's a true cross compiler, and it's hosted on a wide, wide variety of host machines. But it has a limited set of targets, namely, the standard 16-bit machine.

S. Crocker: I was looking for some standard Navy systems that are programmed in CMS-2M, and had expected that NAVMACS, for example, would be one of those that I might find in CMS-2M. I was told that although it was a communication system, although it was mandated to be written under 2M. in fact it was written under assembly language. Where is the trouble? Why the discrepancy?

J. Cooper: I don't dig the discrepancy.

S. Crocker: The Navy has standard languages and has mandated that they be used in development of certain systems, and now we find that particular systems didn't develop that way.

J. Cooper: The language was available and was a standard. What we did not have was a mandate that made everybody use it. The only competitors in the Navy we have are assembly languages. We don't have any programs written in FORTRAN or PL/I or anything like that. DOD instructions 5000.29 and 5000.31 were the first mandates that gave us leverage enough to knock-off the use of assembly language. So now even assembly language usage requires waivering.

S. Crocker: But why was the decision made that way? When?

J. Cooper: Something that DOD has got to recognize eventually is that the Department of Defense is perverted for efficiency, especially for Avionics systems because their space is so limited. Communications thinks they have to be super-efficient. Every community has a reason for its perversion which is extreme.

- S. Crocker: Is there some evidence that these languages can't compile as efficiently as assembly code?
- J. Cooper: There is some evidence, yes. It's biased evidence. They always point out how inefficient the compiler is compared to assembly language. They always compare a perfect program in assembly language, not your average assembly language program.
- S. Crocker: For other languages it is fairly clear that one can compete well with assembly coding on the average.
- J. Cooper: We've done some benchmarking. We know kind of where we stand. CMS-2 is not nearly as efficient as we would like it to be, and it is being optimized right now. On the other hand, for SPI/1, our newer language, we've done three benchmarks, and it comes out around 7%, and that includes all the runtime support overhead that goes with it.
- : In space or time?
- J. Cooper: Both. In these benchmarks we actually did the same program both ways for comparison and came up with approximately the same figure all three times.
- C. McGowan: That is a remarkable figure, but can you say something about what percentage of software development efforts get exceptions and do assembly language implementation rather than use one of the higher-level languages?
- J. Cooper: Since DOD instruction 5000.31 was signed in November 1977, there has been no waiver granted for assembly language use. If anybody is doing it since that date, they are doing so in violation of that directive.
- C. McGowan: Have there been any requests? How many software development efforts preceded that and what kind of numbers are we talking about?
- J. Cooper: Whether it is a DOD or a Navy policy, you never make them retroactive. So the only ones that would be candidates to come under the thumb of this new requirement would be those new starts after that date. There are literally thousands of Navy projects using computers that started before that. It's been a year and a half, and in that time there have probably been a hundred or so new starts.
- M. Wolfe: In the language CMS-2, do you allow embedded assembly code?
- J. Cooper: Have to.
- M. Wolfe: Do you count the use of the embedded assembly language coding as using the high level language?
- J. Cooper: There is no ruling printed on that. In the legal profession they have some court cases that help set the precedence. Clearly, a guy who goes in with his first card

as a header card to drop to assembly language, and the last card in his deck is an "end assembly language" card, you know that he's cheating. In the case of the contracts like the F-18 at McDonald Douglas, we came with a hard percentage. Once you force them into the higher-level language box, then they don't get so carried away with the assembly language. Then they usually use it for I/O and optimization.

- J. Cooper: The real purpose of the workshop is to develop inputs for guiding the future of DOD1. I would like to use the second half of our session this morning to start discussing more of how DOD1 should be managed. For example, should we manage it the way the Navy does CMS-2 or should we manage it the way the Air Force does JOVIAL, or something in between the two, or one of each? These are the kinds of things that would provide good inputs for the Pebbleman document.

To give you the benefit of some of the problems that we encountered at HOLWG meetings I can bring up some of the issues that were raised there. You have to be very careful that you separate in your mind and in your requirements the difference between a language and an implementation of the language. Is that a requirement on the language or is that a requirement on the compiler. You often get fouled up in the two when you fail to keep them separated. Another thing is that there are three levels of control for DOD1. One is that DOD1 has world-wide or international implications and so we end up developing a language to be used by everyone, a lot of people in the world, who we don't have any control over. There is another level, the DOD-wide level of control where we do have the say in how things are done. But even in that context there is yet a third level, that the individual services may do things differently. So as you address the different control measures and mechanisms, you also have to keep the different levels in mind.

-----: Could you expand and enlighten us a little bit more about the two approaches? Particularly, what are the inadequacies of the Navy's approach for doing what it is supposed to do, and what does Sam feel the inadequacies are for the Air Force's approach.

- J. Cooper: The main reason ours works is because we have such a limited set of architectures to target for, hosting's no problem. The more targets that you try to support, the larger it makes the project.

Also, it is almost the same difference as between centralization and decentralization. Correct me if I am wrong Sam, but their approach is more of "here is a specification for a language; you build the language to this spec and then we'll test it." In our case we GFE it. In DOD we have some leverage now. There is a DOD instruction being staffed now that will limit the architectures in DOD.

- T. Cheatham: An architecture is the machine?

J. Cooper: No. DOD is coming to accept a weird definition of computer architecture -- that as seen by the assembly level programmer. It is not a physical structure. It's a virtual naked machine, no operating system.

S. DiNitto: Basically the Air Force does not really have a serious hardware standard. The past history has been that the system development office does not even specify the hardware. It specifies capability, and this puts the weight on the shoulders of the contractor. It leaves the systems program office out of the position whereby they could be accused of putting on unrealistic requirements in case of nondelivery. A wide variety of hardware causes problems because compilers aren't always available.

[DiNitto explains that from his experience, contractors tend to want to use stable languages which have been in use for some time. In the Air Force standardization system, waivers to use non-standard languages are often granted on the basis that the language to be used is very stable.

DiNitto also briefly describes the British Air Force's efforts to control their high-level language by freezing it.]

M. Wolfe: I feel trapped when you say a language is frozen. Language is going to be evolving. It's going to change. You have to allow mechanisms in your control so you can permit this.

P. Elzer: I don't know whether to agree or disagree, but I don't think that it is a law of nature that languages keep changing. Why is it not possible to freeze it for, say, five years with no changes at all? And then do all the changes which may turn out to be necessary after careful consideration all at one time. From a user's point of view, I would prefer such a policy very much to a situation where the standard keeps changing "just a little bit," but all the time.

P. Wegner: The whole philosophy of software methodology is that we have to design systems with change. I think it will be a disaster to design this language and propose that it does not change. Everything does change. The height of those actions is to assume that man is not going to change. A language is a complex application system.

A. Gargaro: I'd like to go back to what Cdr. Cooper had said earlier, that we should be looking at some of the contexts that might help us in language standardization. In my experience, one of the problems with language standardization is that we're not sure what we are trying to standardize. I think it would be very beneficial if with DOD1, we did strive to get a formal rigorous specification of this language. How are we going to do this, and how can we look at the design specification of DOD1 so that language standardization does become a realistic goal, not necessarily wanting to preclude the possibility that the

language will have to change in time?

[Tape breakage causes loss of further discussion on formal language definitions. Conversation resumes with P. Wegners next remarks.]

- P. Wegner: Clearly, what we want to do is to come up with some sort of proposal for language control that will be simple to learn from the current method of language control in the DOD. I don't know how many people you have in your control group; it sounds like a very complicated process. What is needed is some sort of proposal to control the language and also the kind of standards that the control group needs in order to operate efficiently
- J. Cooper: The size of the control groups is dictated by the method. Our control group is very small. I don't know what the Air Force's is. You've got a lot more boxes on your viewgraph than I would have on mine. [Reference is to viewgraph used by S. DiNitto.]
- C. McGowan: Let's suppose that the common language works and becomes one of the standards. Suppose this standardization is adopted. How would that impact your efforts in your agencies, and if it would adversely affect them, what would you like to see change? How would you like to see this session on language support written? How do you envision interacting with your group?
- J. Cooper: Well, I'm in a position where I can take it or leave it. I don't mean that facetiously. For example, whatever is there, I can take it and run it exactly the current Navy way. I can pick one compiler, one implementation of DOD1, freeze it, and say "that's the Navy's standard".
- C. McGowan: But there will be one agency spanning all the services rather than each service having their own. Is that correct?
- J. Cooper: That's not clear.
- S. DiNitto: The point has been brought up that each one of the services would be responsible for each of those three boxes, and I do not agree with that. I think it should be totally centralized. [The three boxes to which DiNitto refers on his viewgraph indicate the areas of language control, language support, and language validation.]
- J. Cooper: I think it should be centralized too. I still feel strongly that the language can be frozen.
- C. McGowan: Do you have any experience in centralized languages? Are FORTRAN, COBOL centralized or are they distributed?
- J. Cooper: No, they aren't centralized at all. I don't think so, especially COBOL.
- C. McGowan: I mean COBOL within the Navy.

- J. Cooper: No. The Navy just runs the COBOL validation for the government.
- C. McGowan: Suppose the common language is as good as everyone hopes. Do you envision that within the Navy, people will be using CMS-2 as much as or more than the common language by 1985?
- J. Cooper: It will depend on how simple it is, how efficient it is, and those kinds of things as to whether it is going to be popular with the user community. If contractors or programmers who want to use DOD1 convince their management they want to use it, then it will ultimately be accepted. You have also got to take into consideration that we are still going to have those seven languages on the currently approved list in the inventory in the year 2000.
- P. Wegner: They might go away in terms of new starts.
- J. Cooper: Yes, new starts, but we will still be supporting those other languages for the next 20 years.
- C. McGowan: Is it true that you and Sam agree that there should be a centralized control that spans the services? That would seem to be a strong statement that's not that strongly placed in the Pebbleman.
- J. Cooper: I agree.
- C. McGowan: Can we discuss the pros and cons [of centralization versus non-centralization]. There must be an opposing team.
- S. DiNitto: I suppose to keep everybody happy, you give them a piece of the action.
- P. Wegner: Are there any technical reasons for having separate control organizations that you can see?
- J. Cooper: Technical, no. They are all political, financial, and other things. Mostly NIH. Everybody's got to have their own piece of the action to feel important.
- D. Luckham: How about setting it up as a DOD organization?
- J. Cooper: I think that is the only way it would work. What was originally in Pebbleman was the implication that one service would be assigned a language control facility, and another service would be assigned to language support, and another service would be assigned to language validation, and they'd divie it up that way. But I don't even think that would work anymore ... With each of those joint projects you end up with three subsets -- Navy, Army, and Air Force. The problem with that is that it's a tri-service system. That is, you have representatives from the services as opposed to having it, not as a tri-service, but DOD, where you have a separate entity, identified with the DOD level, and you staff it at the DOD level.
- J. Bladen: We in the Air Force Armament Lab are under the

restraints of RADC and the JOVIAL control. However, we are also looking at the way the Navy does things in that we plan to have a standard compiler. We have what we feel is a technical solution to the problem. We have come up with a way of standardizing using JOVIAL. What we are going to do is have one compiler. That compiler will be approved by RADC. As a matter of fact, RADC is writing it. Once we get the compiler on the CDC 6600 we are going to use it as a standard compiler, and any new code generators written for any machine will run on that particular compiler.

The next step will be to write numerous code generators to retarget the compiler to the 16-bit microcomputers that we are involved in. So by having one compiler with a library of code generators, anyone of them can be swapped in and out at any time to retarget to a different machine. We achieve total flexibility. Suppose we buy a new computer and target to it, then we use it in an embedded system and we write JOVIAL software for it. If in the next phase we realize that there is a requirement for a faster computer with a different architecture, instead of throwing out all of the software, we'll swap in a new code generator and target to the new machine. The software has not been changed in any way. This does involve a standard compiler. I would like to see this plan put into the DOD1 program.

[Cheatham, Bladen and others discuss the merits of Bladen's proposal. Cheatham points out that adopting such a method for compiler standardization shifts much of the burden of compiler production and validation to the code generator.]

No firm conclusions are reached and the line of discussion concludes with the following remarks by P. Wegner.]

P. Wegner: I think that your problem is a technical one that is very significant for language control. As far as the conference is concerned, we should prepare a statement of the problem with what we see the solution to be, and identify several of these technical problems -- perhaps of language control -- and how they impact the language control process.

[The general topic of discussion moves to the relationship between language standardization and validation. Several individuals suggest that there are important relationships between the two areas. Cooper indicates that another separate conference session is devoted entirely to the issue of validation.]

The following remarks by Luckham and Cooper summarize the remaining scattered discussion.]

D. Luckham: I'm sure these topics about validating and standard compilers will come up again, but I read this paragraph 3.1 of the PEBBLEMAN as saying to minimize changes to the language. Now, if anyone has been through

the four preliminary language designs, you can see an abundance of new constructs. I think it is very unlikely that by 1980 there will not be any new good ideas that have not been incorporated in the languages -- some obvious, provably good changes. It's quite clear that you are going to get an evolution in this language whether you like it not. So what is your procedure going to be when somebody comes up with a reasonable suggestion?

- J. Cooper: Don't have any answers. That is what we are trying to generate with Pebbleman.
- D. Luckham: Then I would suggest that you'd better have a board of experts, either killing the idea by showing how the code is just as well without it, or else how to compile it and update the compiler. Somewhere here there has got to be the technical expert advising.
- J. Cooper: We have had lots of interest in tools. In fact 99% of the interest in Pebbleman so far has been in tools. The second most popular area is certification. Nobody seems interested in the things that we were supposed to talk about here in this session -- Sections 2, 3, 4 and so on. Yet the policy decisions that are made in Sections 2, 3 and 4 are going to dictate what you do in tools and validation, etc. They set the framework for standard intermediate languages which you are going to have to have. They set the stage for whether you even have a root compiler or not. So if you feel strongly about having a root compiler or standard intermediate language, you really ought to provide inputs to the other management and control sort of subjects.

Session 2A: Technology for Language Specification
Steve Crocker, Chair

- S. Crocker: In Section 5.2 [of the preliminary PEBBLEMAN document], under compiler validation, the second paragraph says: "The method of validation should be to compile and execute a standard series of programs written in the common language to test for correct translation." This is not strong enough. There has to be some provision for much stronger types of analysis of the compiler and determination of the compiler coverage of the language.

The most important step that can be taken in this area is to insist on a formal and rigorous semantic definition of the language. We all know that the techniques for formal semantic specification of a language are not well developed, and the few serious attempts have encountered various troubles. But I am convinced that the attempts must be made anyway. I listed several reasons: [Crocker refers to a prepared viewgraph.]

A formal semantic definition is a prerequisite for formal verification. Even if the formal definition is hard to read, a sufficient number of people will read it and understand it. In particular, compiler writers will understand it and base their implementation on such understanding. It is quite likely that we'll learn how to write readable -- even pleasing -- formal semantic definitions. If we do learn how to write these things and they are readable, they may become the reference of choice for both more users and compiler writers alike.

Finally, some reference documents are absolutely required, and these reference documents must serve as the basis for arbitration of differences of understanding that undoubtedly will arise. So even if the English or other formal documents were more readable, the formal documents provide a better chance for eliminating ambiguities.

The focus of this section will be on: what are the prospects for formal semantic definitions, how can we get one, what the tools are, and what tools might relate to this activity in the common high ordered language efforts.

- D. Luckham: We'll just take a look at some of the techniques now available for formally defining programming languages. The first technique we'll look at is VDL, which is the Vienna Definition Language. VDL was developed by IBM Vienna laboratories to provide a formal definition for PL/I. VDL is based on a concept of an abstract machine. The methodology goes as follows: you take a source program and translate it by means of an algorithm called a 'translator' into an abstract program; this abstract program is then executed on an abstract machine by means of an algorithm called an 'interpreter.' The meaning of a program is defined as the sequence of changes in the state of the machine as that program is being executed.

One of the advantages of VDL is that you can provide detailed information about the language you are defining using this technique, and the abstract machine is an intuitive method of demonstrating how the language works. However, it is a nontrivial method to understand, and using this abstract machine you might bring in extraneous detail that could obscure some of the constructs of the language. Also, the translator/interpreter mechanism is not necessarily a distinct division in a programming language at all.

Another technique which was mentioned earlier is to use W-grammars. W-grammars are two-level grammars developed by van Wijngaarden. They were used to formally define ALGOL/68. W-grammars are not easy to understand. It is possible to generate an infinite set of context free productions by combining two sets of rules: hyper-rules and meta-productions. The combination of these two rules generatively defines the set of legal programs in the language.

S. Gerhart: How do you know what a program means in W-grammars?

D. Luckham: In the first phase, W-grammars can specify whether a program is correct (i.e. legal) or not. The W-grammar itself generates the set of all legal programs in the language. To find out if a program is correct you have to follow the grammar itself to see if that program could have been generated by the W-grammar.

S. Crocker: The distinction is whether or not you have a valid program, the one accepted by the definition of the language, versus whether you know that it executes the way you expect it to execute -- the summary statement of what your expectation is. How do W-grammars specify what the output is supposed to be from execution of the program? Do they do that?

D. Luckham: Yes, they do. I don't know that I can really answer that question.

[Luckham gives some sketchy details of W-grammar meta-notions and hyper-notions, productions and rules.]

P. Wegner: This is the method for justifying the syntax not the interpreter. There is no interpreter associated with it. The interpretation mechanism as a separate thing in the ALGOL 68 report is given informally. It discusses how things are elaborated.

D. Luckham: However, there are semantic issues that are taken up in the definition of the language.

P. Wegner: Yes, for example the language takes care of the relation between declaration and use of variables, if you refer to those as semantic issues. But the actual execution of the interpreter part of a VDL definition is not part of the W-grammar formalism.

Using a W-grammar it is basically possible to do any Turing machine computation, so you could specify anything you wanted. However, the way they're used in the ALGOL 68 report is to specify the syntax down to excluding multiple definitions of a variable in a blockhead, and making sure that variables you defined are also used, and things like that. It specifies syntax down to a finer level than in the ALGOL 60 report, but it does not handle interpretation at all, although in principle it is possible.

- D. Luckham: Some definitions have been created that do have the interpretation. The ALGOL 68 definition may not, but it has been done with W-grammars. The only advantage I see is with the single formalism, you can understand what they're going about. However, the technique is entirely generative and you have to follow through all these rules to see if a program is legal. It's not necessarily the case that you can see if a program could not be generated in a language, and if there's no isolation of the context sensitive requirements from the context free requirements in the semantics, it's very hard to read.

[Luckham resumes his discussion of the various semantic definition techniques.]

The third method is using attribute grammars. This is a definitional technique was developed originally by Knuth and used to formally define EULER. It is a context free grammar where you associate the attributes with the nodes on the derivation tree of grammar. Some set of attribute evaluation rules are associated with all the productions or semantic functions. One of the advantages of attribute grammars is that they're easily understandable. However, they're not a full technique because you must combine some additional formulas to define the semantics of a language, like denotational axioms, axiomatic semantics, or semantic functions of some type. Some alternate methods include production systems and Semanol, which will be discussed later.

- S. Crocker: One of the questions to keep in mind is: "How much experience has there been with any of these techniques and what are the prospects for actually using formal semantic definition systems in a practical way?"
- P. Wegner: One related item is there's been a PASCAL definition which is not totally formal, and then EUCLID, and then various other definitions along those lines. The IRONMAN and STEELMAN are oriented towards that style of definition. It's quite true that the PASCAL definition, as such, is incomplete and leaves lots of holes, but it may well be that the direction to go is to tighten up something like the PASCAL definition and make it semi-formal (more formal than it is now). That might be the style of definition more appropriate to DOD1.

[E. Nelson now gives a detailed presentation of the Semanol system which is described in his position paper. He concludes his presentation with the following

proposal.]

I propose that a standard definition would be comprised of four elements: a Semanot specification, an axiomatic specification provided by the contractor, a reference manual defining the language in English, consistent with the other two, and the compiler validation test cases.

W. Loper: I have a question about forms of parallelism.

E. Nelson: The DOD common language requirements requires handling parallelism. These are admittedly new types of language features which are not present in previous designs. Looking into things like monitors and boxes, as described in these preliminary designs, it would appear that Semanot has the facilities to deal with them. It might turn out that you would want to define some new high-level concept in there to make it easier to describe, and there will have to be some work in detailed modeling of what the language designers actually produce. We believe it is a solvable problem.

S. Gerhart: How about a fifth component to your standardization? A proof of consistency between the axiomatic and Semanot specifications.

E. Nelson: Yes. I'm not sure whether we know how we'll find complete formal consistency, but there are probably a number of tests that can be applied to check the consistencies of these two definitions.

S. Crocker: Why are two formal definitions required?

E. Nelson: In principle, you would need only one. The axiomatic specification which is being produced is not adequate because it is incomplete. It only covers a portion of the language. Having the two of them we think, particularly in a language as important as this, does help in checking the consistency and provides a different way of testing and using it. We believe the axiomatic definition, although incomplete, is useful in itself, one reason being its relation to current formal verification methodology.

S. Crocker: This leaves me a little uneasy if you say it's incomplete but consistent with the executable Semanot. What can you verify if it is incomplete? Why can't you derive those axioms?

E. Nelson: The incompleteness of the axiomatic specification does mean that the formal verification technology is not as solidly grounded as people imply that it is. It means that they are dealing at best with some kind of partial proofs of correctness because they have incompletely covered all definitions. That doesn't mean it isn't useful.

S. Gerhart: In fact, the axiomatic specification is only a part. There's also the identification of datatypes, considered as somewhat of a separate definitional mechanism. It's more than two definitions that you're talking about.

Operational versus some other kind seems to be the main distinction.

- P. Wegner: Which definition would you go to if you had to decide whether a certain language feature was correct? The Semanol definition?
- E. Nelson: I probably would do to Semanol because I'm more familiar with it.
- S. Crocker: What do you mean when you say that a language construct is correct? Do you mean the compiler is right, or do you mean the user was using the language feature correctly. There has to be one meaning, there can't be several.
- E. Nelson: If they disagree, then of course they are inconsistent and you may want to resolve the inconsistency.
- P. Wegner: If we have these several layers of definition, then what will happen in practice is that, and this is good, when people have a problem concerning the language they will probably go to the English definition first. And if it can be answered at that level, fine, and mostly it will be able to. Hopefully, one of the reasons for having the more formal definition is to get the English definition right. And, hopefully the English definition will be sufficient for most purposes. In practice, the English definition is going to be crucial to everything, and that's why I prefer it in the PASCAL style probably. What we're really working at is to get a good and complete English definition. In a sense, the formal Semanol definition and the axiomatic definition are, too, to get a better English definition. ...

One further point concerning your comment and discussion: what you're really saying is that in the class of things that are more like operational definitions, you include Semanol and VDL and so on. Semanol may well be the best way to go, as far as that is concerned. The other point is that several complementary definitions are probably the way to go.

- S. Crocker: A couple of questions about the experience you've had with Semanol. How big do these definitions turn out to be, how much effort do they take? How come it hasn't spread like wildfire across the landscape?
- E. Nelson: Here is a copy of the JOVIAL J3 Semanol specification. It is not very densely packed. [Nelson displays a document of a dozen or so pages.]
- S. Crocker: How big is the English spec?
- Col. Whitaker: The natural language definition from which the Semanol specification was developed is about six or seven pages. It is smaller print and it is double columns.

I would like to back up what Dr. Nelson has said. We have found it [the Semanol definition] to be a very useful tool

in debugging the language specification itself. It kind of backs up what Peter was saying. In talking to the British about their experiences with Coral 66, they did undertake a formal definition of that language. The problem they ran into was that the people they had to approve it couldn't read it. I think we do have to stick with the natural language definition. It's the standard as far as we can take it because right now, I think, the state-of-the-art, even Semanol, is a lot of work to read. If you want to look up something like the example he gave, you want to find out exactly how loops work, for example, you have to look somewhere else which in turn will refer you to something else. The information is there, but it is a lot of work to get it out.

- S. Crocker: What are the prospects for bringing those two things into conjunction so that the formal spec is a readable, even pleasing, reference document?
- E. Nelson: I would say you'd have to start with the natural language specification, then work to the formal specification. Where problems are encountered, well it's going to be a matter of editing natural language specifications and resolving the problem where it's identified.
- S. Crocker: I'd like to provoke some discussion on the point that Peter raised about a semi-formal specification being the most natural resting point. I'm fairly enamored of the idea of being able to take a formal definition and do several things with it. Execute it for one thing, just to see what candidate programs are going to do. Input to formal verification systems at one point is another kind of thing. Using it as a top-level spec for compiler development, either as a target for verification of an implementation of a compiler, or as a starting point in automatic design of a compiler by successive transformations. Contrary points of view?
- P. Wegner: I think that maybe we ought to take the position that the English definition should be the first recourse and possibly even the final arbiter. That's not to say that a formal definition isn't very useful. It should even be required and used for testing our programs. I go with the idea that you have both and require both. The English definition is the arbiter and the formal definition is there for validating things as well.
- W. Loper: I don't think it's enough. I've had two experiences of being among the first to implement a compiler for a newly defined language. In one case, we had to implement FORTRAN IV to obey a public standard, not an IBM standard. In the second case, I was perhaps among the first to try to implement PL/1. In implementing PL/1, even after they had completely written their final language specification documents, that was so far from supplying guidance to the implementation that I have a section of a filing cabinet devoted to the subsequent correspondence to find out what in the world they meant when they had written what would

normally be accepted as a complete and finished language specification.

Later it became easier (this had been back in 1966) because the questions had been settled. Two reasons: one, you had a tradition. Everyone knew by that time that when a language said X it really meant Y, so there was no problem. There are things that in reading the FORTRAN definition you stumble over without realizing that it simply doesn't settle the issues. You have to know from long experience how the issues have been settled.

- E. Nelson: I'd like to talk on both sides of the question. On the one hand, this ambiguity in the English is a very real problem. As you said, the way English specs are usually written probably would not provide guidance to many issues. If you do an iteration, starting from the English definition to a formal definition which is precise, then go back and rewrite the English definition, now knowing exactly what it means, then you can remove a lot of ambiguity you couldn't resolve in the first place in English. English being itself not a formally defined language, you may still not be free of ambiguities. While it may be the arbiter of most cases, there might be ones where you'd have to go back and read the formal spec and say this is what it really precisely does.
- P. Wegner: I think I now agree. The formal definition should be the arbiter, but that it's understood that the English definition is as complete as possible and that in 95% of the cases, that will be the one that is accepted.
- V. Sneider: My own opinion is that any definition of a programming language ought to be a top down definition. Where you look at the very top-most level of the semantics and use highly abbreviated and symbolic notation at the top which conveys what's happening without giving the details. If you want the details, then you look it up in the manual under the section which gives the specific actions that are going on at this level of the translation or interpretation process. That's not inconsistent with using English at the top most level and referring you to details with this or that section of the further report. I don't see why a person is supposed to comprehend everything that's going on from top to bottom in the translation process when he just wants to get some vague idea of where things are headed.
- S. Crocker: I think you're raising a subtle but extremely important issue that pervades all requirements specifications. That has to do with how you focus on essential or normative cases as opposed to all of the myriad of details, many parts of which have to do with boundary cases and error conditions which are not what a user wants to find out about upon first reading. It would seem to me that one of the things we trip across when we try to write formal definitions of a compiler or languages is we haven't found a way to bring out the core of what we're trying to specify in an easy to read and focused way, and still have connection with all the myriad of details that must go into a full formal specification. Does anybody want to talk

about the SRI work? It is the only work I know of that mentions that kind of issue.

- A. Marmor-Squires: Are you talking about "Special?" [Yes.] Special, as I understand it, is not purposely designed to define programming languages, but to provide a high-level non-procedural definition of programming.
- S. Crocker: True, and I meant to point to it only as an example of a specification language which has as its attribute that it separates normal behavior from error behavior, not that it's a language specification system. That's an attribute (separation) that we may want to have in a formal semantic definition system for languages, which would speak to some of these issues about how big the spec is and how much time it takes to read it and who's going to read it, who's going to understand it, and hence, what its ultimate effect is going to be on the community at large.
- P. Wegner: Suppose we choose a certain definition style. We can still write good and bad definitions in that style, and one of the things that Victor was referring to is that the definition should be structured in some way, top down, for example. I'm sure that there is a great deal to be learned in writing definitions well in any notation we care to use.
- S. Crocker: How much experience has there been with formal definitions?
- V. Schneider: I don't know anybody who really has an absolute, water-tight formal specification of what a compiler is doing and what its runtime support system is doing at the same time. That may be too strong, and it may not even be humanly possible. There might be someone who could prove that it really isn't possible to give a water-tight complete definition. The question then is, do you want perfection or do you want something people can use? What is it that people can use? I'm proposing the top down structure definition of the language.
- S. Crocker: One of the driving forces is whether you can move software that's written in a language from one machine to another machine after you recompile it. How much damage have you done? Two kinds of answers: either you're prepared for no damage at all, in which case you have to have a water-tight system; or you have to be prepared for some and then it's a question of how much, and we get into the usual hagggle. So you have to go back through some testing and validation. That comes back to some policy issues about whether we're going to manage our way out of it or whether we're going to have some guarantees. So it's a question of how close we can come and if it's close enough.
- D. Luckham: A small comment on the amount of experience with formal definitions. There is no experience with formal definition of parallel processing. All parallel processing languages I've seen have no formal definition.
- P. Wegner: It would seem to me that on concurrent processing

there are no insuperable difficulties of extending operational techniques to concurrent processing. A little bit extra will have to be done with synchronization. Some extension of axiomatic techniques has been done.

- D. Luckham: I'm not saying that there are any insuperable difficulties, I'm saying that the experience I've seen is zero.
- S. Crocker: How much experience has there been with validating compiler construction?
- R. Morris: It's alleged that COBOL's fully validated, but I don't know anything about how it works. Col. Whitaker mentioned this morning that the COBOL people have done it.
- Col. Whitaker: The compiler construction is not validated. The compiler is validated by some 350 plus test programs, specifically designed for that purpose.
- E. Nelson: Having had this discussion concerning COBOL and its validation system, there continue to be many arguments over the ambiguities of interpretation of correspondence on that area. It isn't 100% decided, there are other things, such as a Semanol specification of COBOL, as a step toward a better standard.
- P. Wegner: Are you looking for a Semanol specification for DOD1 languages?
- E. Nelson: We've been doing some studies on it.
- S. DiNitto: We've talked to our lab director about sponsoring a Semanol definition of DOD1. We can't go the full route. I doubt if we'll have the funds to do the build-up of the translator and interpreter so that it will handle DOD1. But we still think it will be valuable to debug the specification just by undertaking the definition.
- S. Crocker: There's a small subtlety which you've just raised about building up the translator and interpreter to handle the extensions to the Semanol language to handle DOD1.
- E. Nelson: In resolving the questions relative to specifications there's a question of how much one wants to describe in high-level terms and how much in low-level, which compounds the detail in there. For parallelism and certain other advanced features, there may be for readability and execution purposes some operators in the language that directly mean this rather than describing it in terms of how you manipulate strings.
- S. Crocker: Give some further indication of how much extension to Semanol you'd find useful in describing a language as rich as DOD1.
- E. Nelson: I don't think there'd be very much of an extension.
- S. Crocker: How many people here are involved in compiler

construction in one way or another? [Hands are raised.]
About a quarter, a third.

- E. Nelson: With this number of compiler writers here I'm surprised that there hasn't been raised the usual compiler writers complaint that formal definitions over-constrain them.
- A. Gargaro: I have a question for Dr. Nelson. Has TRW looked at Semanol for defining COBOL?
- E. Nelson: People have looked at it, and believe it is do-able, though it has not been done. If you talk to us a year from now we may have it done. The various data structures, table structures, picture-spec, etc., are different than in other languages. They don't seem to raise any insuperable questions in describing them in the right amount of detail and reaching agreement on what they mean.
- S. Gerhart: Is there sufficient mathematical theory to support any sort of formal definition of parallel processing? Over the years a theory has built up that at least makes the definition more believable.
- D. Luckham: First of all, I don't know the answer to that, to be honest with you. I can tell you what I think at the moment, which is not necessarily going to be true tomorrow. Yes, I think there is enough mathematical technique to develop a specification language for parallel processing. The first thing that has to be decided is what people want to say about the processes. It seems to me that the old I/O specs that would do for procedures no longer do for parallel processes. Naturally, when you get into a language in which you're talking about infinite streams of flow of information, the mathematics of that is problematic, but solvable. If you have two processes accessing the same input channel then you get into the mathematics of shuffling operations or subsequences, though you don't know in what order the accesses will occur. The mathematics of that is solvable although in some studies it was ignored.

There is a second line of problems which has to do with synchronization, which is separable from the flow of information. Most well-written operating systems tackle the problem. The formalization of synchronization problems may depend on very precise programming techniques. My feeling about the question is, yes, the mathematical techniques to solve the problems exist. It's a question of what people will accept as standard specifications for processes. Or we can decide what languages we'll use as a standard.

- P. Wegner: This is in the area in which you're working? [Right.] So you are in fact developing techniques for specifying modules and concurrent processes.
- D. Luckham: What I would claim to be able to produce in the near future is something that would be adequate for very, very simple kinds of processes, the ones you would see in a simple kind of operating system. It is not for the sort of

processes that might evaluate a numerical analysis problem in some very efficient way.

P. Wegner: How do you feel about axiomatic definition of DON1? How complete could it be within a two or three year time period?

D. Luckham: I don't know. No. The designs weren't specific, you know. I can tell you where the problems would be, or some of the problems I don't know how to solve. I don't have a full enough picture to give you a complete axiomatic definition.

S. Crocker: From some of the other compiler writers, what is your preference for specification of a new language?

-----: Clarity. You spend more time on trying to figure out difficult issues that haven't been clearly settled than you do in implementing the ones that have. Only a minority of my time is spent in implementing the things I understood, after I found out a little about them.

S. Crocker: What do you think of the idea of a reference compiler? One whose sole purpose is to provide an operational model of the machine rather than one that's aimed at efficiency or target code production. Perhaps a reference interpreter is the right idea?

D. Luckham: It would not be extremely valuable. I would much rather do a program. It's quite clear that if it's written down, you can follow the source code of the compiler or even the bit pattern that is executed.

E. Nelson: In the development of the University CMS-2, they were developing a compiler at the same time we were writing the Semanot specification, and we had meetings to resolve issues and found those most productive. The compiler writers managed to illuminate several issues for us. We resolved questions so that what they wrote down on the compiler spec and what we wrote on the Semanot spec were consistent.

D. Luckham: I'm curious about the reference compiler. We're talking about some formal definition which hopefully is as unambiguous as you can have. Of the methods of definition, I would think Semanot could write answers in the same way as any particular program could provide answers. The idea that a man would go to a piece of paper as you referred to it, is usually a lot more practical than running through a particular compiler you happen to have. Having the compiler should be more of a solution than having a formal definition that has been agreed upon. Hopefully, you can look at that formal definition and imply the answers.

S. Crocker: Why even use Semanot when you have both?

D. Luckham: That's a good point. If the technique isn't able to be executed, you may have a more difficult time proving that it is clear.

- S. Crocker: So one might see Semanol as a language for writing language compilers.
- E. Nelson: What a formal definition can really be looked at is the compiler that doesn't provide you with any extra information. That is, compiling for a machine or in an environment that discusses only those semantic issues you want to address.
- P. Wegner: It's an evaluator, really, rather than a compiler. The compiling aspect is somewhat irrelevant.
- : How do you pose a question to the reference compiler?
- P. Wegner: You pose questions about what a program does rather than about what a program compiles into.
- S. Crocker: So you really want the reference interpreter?
- A. Evans: VDL does that. Not so usefully perhaps, but VDL provides you that mechanism.
- S. Crocker: What is it that makes VDL not so useful?
- A. Evans: It takes a long time to answer any reasonably hard problem. To submit a piece of program to VDL and work it through by hand would be a very time consuming operation.
- P. Wegner: I think VDL and Semanol are competitors for a formal definition mechanism. We could go with Semanol rather than VDL because Semanol is more recent. Are there any other competitors for this kind of definition? Namely, an interpreter which is also implemented so you could run things through it. If VDL were implemented you could run things through VDL.
- E. Nelson: The problem with VDL is that they have the different sections on what they call the concrete syntax, abstract representation of the concrete syntax, the abstract syntax and the abstract machine. They tend to be of quite different notation and in most cases people have not done all of the sections of VDL. Most usually leave out the abstract representation of the concrete syntax. Also, the abstract machine is unlike the Semanol interpreter. It actually holds part of the language definition and so you have a different abstract machine for every language.
- D. Luckham: I just want to make a point about compilation versus evaluation. We've been doing some stuff with attribute grammars. With that point of view, you're compiling a mathematical function, if you look at it as a function of the top node of the program. I think we should make clear whether we're trying to come up with an abstract machine as a series of state transformations and if we're really trying to say something about these particular states or whether we're considering the procedure as a mathematical function.
- E. Nelson: I think that latter point is important and also one

of the features that Semanol is based on, a theory of semantics, whereas, VDL gives the ultimate interpretation on how it executes in terms of the machine. The programmer is concerned with describing an information problem and the programming language ought to have something to do with describing information problems.

Session 3A: Verification Technology-Present and Future
David Luckham, Chair

D. Luckham: The first thing I want to say is that there seems to be a strong image of verifiers as a black box into which the programmer will put a 20,000 page FORTRAN listing with just an I/O specification and out will come "true". Such a black box will never happen. My view of verification involves many of the aspects that will be going on in the other sessions. Programming language design is an intimate, important part, as is the design of specification languages and the design of a methodology for documentation.

[Luckham refers now to an overhead projector slide containing five points which illustrate the relationship of verification technology to some important related areas.]

We need to give the programmer the following items: 1) the tools to write a program, 2) the tools to state his intentions, 3) the techniques for stating specifications, 4) the theory of how he will establish consistency between code and specifications, and 5) a system to help him automate certain aspects of the proof.

Euclid is an example of a programming language design specifically oriented towards making the task of validating programs easier. Whether it was a good decision or not, I don't know. The introduction of typing in the language, which came a little earlier, is a perfect example of forcing the programmer to declare his intentions. It turns out that he declares a lot more when he makes his type definitions than is checkable by the standard compilers, and so we can still design verifiers to use that information to check further what is normally checked at runtime. I believe that language design is going to tend towards the direction of more of these non-computation declarations of intentions. I can see that already, for example, in the Green design.

For specification language design, there are a couple of things to note. If you want to talk about a database program for example, you might want to have a language in which you defined a concept like "cycle-free", tree structures of records, the concepts of searching and manipulating such structures, etc. You could then imagine in the specifications of your program that it would be natural for you to write down something like "the tree structure is loop-free." And so one wants to get the programmer a language in which he can talk about his "higher level" concepts. One has to design this. One has to design the means for allowing the program to design its own specification. Another example would be the language of model logic. If one is talking about specification of synchronization problems in operating systems, it becomes convenient to be able to say "if this happens here, then it's necessary at some time later that some other event will

happen"; "if this process signals that then at some future time something else will happen"; "if this process waits for that resource then it will get the signal that that resource is free." There is now an expanding theory of the old tense logics in a new light, in the light of specifications.

Examples of documentation requirements are things like: what are you going to have to say about a global variable? In a lot of programming languages, you don't have to say anything, you just declare it some place at the top and a few blocks later you use it. For a verifier, you have to declare it at the point where you use it as a global. Another example would be: should there be a standard whereby you have to understand what an invariant for a searching loop is? Should you have to provide one for every loop? Should that be a documentation point? These are examples of what I mean by requirements and methods being part of the technology of verification.

Now, I will say a little about my own program; it is sort of a test case. I look at the verifier as just one kind of program analyzer. Its particular property is that it analyzes the consistency of the code with the documentation. I'm making a rather fuzzy distinction between the word specification and documentation. A specification language is one in which the documentation is stated; a program specification is sort of the global external intent of the program. In the case of a procedure it would be the entry/exit assertions. The documentation is the program specification of all the other internal assertions you might make to explain what's going on internally. In other words, if in your language you use the concept "loop-free", then at the current state of rudimentary verifiers you have to explain to the verifier what the definition of "loop-free" is. The definition of the specification language is just the semantics of the concepts in that language. If you were dealing with sort programs, for example, then you would use concepts like "permutation", "ordinate range", "preserved in the range", "greatest element in the range", "least element in the range", etc. You would actually mathematically explain each of those concepts. Then your specification language would be a language in which those concepts exist. You have to give the definition of concepts to the verifier.

In order to do this analysis, we require that a program be documented and we require a definition of the specification. Given all of those things, the outputs from the verifier are either a proof, in some logic, that the code is consistent with its specifications and documentation, or that it's not consistent and you get back some hint as to where the problem is. You also get back the unprogrammed parts of the logical conditions and the trace of where the proof attempt failed. It's this information that is really most important in developing applications of verifiers, because it's this that is going to allow us to decide where the problems lie in the consistency, and which part of the documentation is not adequate or where there might be a bug in the code. So,

one of the things that we have to do in verification technology is develop tools for analyzing the failure of attempts to verify.

At the present time we are at a very elementary stage in this technology. I'm not even sure we have the right logic of programs, but I know that we've got a good enough language of the logic of programs that we can do quite a few things. We might be like the Greeks, with the concept of infinitesimals and no real notation. We need the axiomatic semantics of the programming language as a pre-requisite for building the verifier. We need "correct" definitions of the higher-level concepts in the specification language. This is so, for example, if you wanted to use loop-freeness as a concept, you would define it by means of axioms, and then you would have to go away and think as to whether your definition really meant loop-free. We also require that the program be documented. These are the prerequisites.

- P. Wegner: When you say documentation, you mean something fairly formal? Presumably, the documentation has to be written according to certain rules rather than just free format.
- D. Luckham: Yes, I mean part of the technology is that you have to lay down documentation methods.
- P. Wegner: Could there be a documentation language?
- D. Luckham: There would be a language in which you could define your own specification range. In other words, if you've got a parser, you're going to want to write a different kind of specification from a sorting program.
- P. Wegner: I'm talking about the documentation. If the documentation is required for the verification, then you're going to impose some pretty stringent requirements on what form the documentation can take.
- D. Luckham: I'm going to negotiate with you. If you're a programmer and you don't like my requirements then I'm going to try to program my verifier around them. It's up to us to decide what we can both live with in the way of documentation methods.
- P. Wegner: But the thing we agree on would be something fairly formal.
- D. Luckham: Yes. For methodology, you must have something here if you expect to get any reasonable answer. So here are some of the uses of verifiers. The first is, in making the documentation more precise, you can't get away with sloppy English and you can't get away with saying "this loop is supposed to do that" and forget all the end conditions, and so on. The first thing that happens when you get into this game is that you have to design rather precise specification languages and documentation methods. I think that is a coming discipline. In the future, there will be much more

rigorous standards for declaring intention.

Having gotten the program, the documentation, the definition of the specification language, and the verifier, the next use is in debugging. That is why failure in verifying gives you information about where the inconsistencies are. If the reason why you didn't get a verification was because the documentation wasn't adequate, then the system forces you to improve your documentation. Also, if you are following a top down methodology of programming, and you've left certain subprocedures external and unspecified, then you'll probably find that verifying the top-level forces you to change or modify the specifications on the unwritten code.

Finally, when you get a verification you get a verification of what you stated. When people say "we've verified an operating system", or "we've verified a compiler", or something like that, what happens here is they have a precise statement of what they verified about a program. They haven't verified a compiler, they've verified that it does some particular kind of transformation. They haven't verified an operating system, they've verified that if blocking and starvation don't happen, then the flow of information from the card reader to the line printer will be as they want it.

I think that nothing will ever be absolutely verified. What you're doing is raising the level of confidence, and you're being rather precise about what you have confidence in. Finally, once you have a verification, that isn't the end of the game. That's just the beginning. Especially if you're working with code that you expect to modify, or specifications that you expect to undergo some modification in the future. You now have a perfect tool for playing with what happens when you make small changes. What we're trying to do here is develop techniques for using and for programming with such a verifier.

R. Balzer: Are you referring to your project or this meeting?

D. Luckham: My project. I'm trying to convince you that this is a much more fruitful way to go. There's been a lot of work on sorting programs, we seem now to have a reasonable specification language with about half a dozen primitive concepts in it. It is satisfactory for documenting and verifying most published sorting algorithms, including versions of heap sort. For pointer manipulation programs we are able to deal with things like the Shore-Waite marking algorithm for garbage collection, various kinds of list processing, single queuing systems.

Back to the type definitions. We have implemented a special version of this verifier which attempts to prove the absence of common runtime errors. What it has to do is build up its own documentation, so it contains an analysis phase which attempts to construct assertions about arithmetical facts. Those arithmetical facts imply the absence of certain kinds of runtime errors and then it attempts to prove the

consistency and if so, the outcome would be you know that you would not get an array subscript going out of bounds at runtime; you would not get the access of an uninitialized variable or a variable with an undefined value; you would not get referencing of a null pointer; you wouldn't get a stack overflow in a recursion; you wouldn't get division by 0. It's like an automatic documenter for those programs for which it is successful. That is, you don't have to supply any documentation at all for the kinds of programs for which you are now still required to supply some documentation.

[Unknown]: What do you mean by an automatic documenter?

D. Luckham: Well, imagine that you just have a task error code and you have no documentation about what the code is supposed to do. What the automatic documenter does is attempt to build up documentation at certain points in the code.

[Unknown]: What does it provide you with?

D. Luckham: It will provide you with assertions to the effect that certain a variable is not zero, or something like that.

J. Prescott: In your formal specification languages, do you find it necessary to come up with a specification language for each type of programming, like sort programming, or do you have one specification language?

D. Luckham: What we have right now is a language in which you can design your own specification. Basically, we have one specification language, but it allows you to define more concepts, to axiomatize more concepts. It's like a predicate logic in which you could define arithmetic or something like that. You could define the theory of sorting, the theory of database management, etc. I don't wish to defend it as anything more than a rather rudimentary specification.

In order to build a system like this, I can name for you some support technology. This has to do with the implementation of special purpose theorem provers and algebraic simplifiers. For example, if you're dealing with a programming language which has standard data structures such as arrays or pointers or files or lists, then you'll probably find that you need to have a special purpose theorem prover for each of these standard data structures. And so there is a support technology in the design of those theorem proving algorithms.

The next thing that happens is you find that each of these special proving boxes have to cooperate very efficiently because they influence each other. Something that the array prover finds out may be of importance to something that the arithmetical prover wants to know, so they have to cooperate and they have to pool their knowledge. This gets you into a concept of cooperating special purpose boxes, and the theory of how you design boxes. Finally, you have the design of

the specification language, which here I've called the proof rule language.

Down at the bottom of all of this are various classical problems that also require solution of some sort for such a verifier to run in real time. We need to improve certain strategies, such as garbage collection. We need to work on printing formats which eliminate common sub-expressions or don't allow them to occur in the first place. Only just last week by going from the MACLISP garbage collection to our own garbage collector, we improved our runtime efficiency, in fact, we doubled it. There's a lot of work to be done on the strategies of when you collect garbage and where you go to look for it.

A few facts about the actual system. It's a PDP-10 MACLISP system of 100K PDP-10 words. There is a user manual we're writing now, and we're attempting to distribute it to a few selected ARPA sites because we think we're now at a stage where we can give it to other users without causing too much ill-will. We would like the feedback. The major problem in transferring it across the ARPA network is the character set compatibility.

[A brief discussion on character set compatibility ensues. Crocker observes that Luckham's problem can probably best be worked out by Luckham.]

- D. Luckham: The language accepted by the verifier at the moment is an extension of PASCAL and includes union types instead of variant records and also modules. We're working on a theory, starting from scratch, writing a compiler and working up the structure it should have so that we can specify it, and verify some properties of it. We're doing this for a mini-PASCAL compiler that includes gotos, iterative loops, procedure calls, block structure, and arrays, but not pointers. Operating system verification is something that I've gotten interested in and it requires a study of all of the problems I mentioned on the first slide, i.e., language design, specification languages for concurrency, etc.

[Unknown]: What about runtime error checking?

- D. Luckham: We're certainly trying to extend that and look at the sort of runtime errors one will get into with, say, union types and module interfaces. I think now, my own evaluation of this project is that we're at the point where we would like to try to do an in-depth study of a PASCAL applications package. I would like a database package. We would like to take a group of between 15 and 50 programs from some place else and see what we can do to develop the specification language for them, develop programming methodology for them, debug them and verify them using the system.

- R. Balzer: You're saying that you believe your system, when operated by the developers, that is people who are experts,

is capable of handling an existing package of PASCAL programs.

D. Luckham: No, I'm not saying we can take existing user programs from other places and drop them in the top and get answers out the bottom. I'm saying that we're about ready to start looking at "can we restructure these programs, develop the specification language, develop the methods of documentation, document the restructured programs, and then debug them?"

R. Balzer: So, in other words, you're saying that it's capable of verifying real applications when they are done with the right technology.

D. Luckham: I'm ready to try doing it.

V. Schneider: I have two topics that you didn't address. One of them is the use of exercisers for verification. In particular, there was a recent paper in the Communications of the ACM on the subject of proving that test cases are sufficient for testing error-freeness of the program. I don't know if you have any comments about that.

Second, there are some theories at the present, I won't call them anything more, about the information theoretic structure of programs. There's one by Gregory Chatin of New York University, for example. Another theory was originated by Maurey Halstead of Purdue University. The theory goes as follows. When you get past a certain point of complexity, you have a chance for an error. If you carry this theory forward, you can predict the number of errors remaining in the program after delivery as a function of the number of errors found during the production process and the size of the program. It seems to me that that is part of the verification process. At least disproving that theory is part of the verification process, or possibly using it in some productive way.

D. Luckham: I believe that the theory, as you have explained it, is quite true. Let me deal with the first point and then the second. I have not read this ACM paper, but some of my students did, and they went so far as to rigorously specify and verify the example programs in that paper in about half an hour, from start to finish using my method. And I think they might even write a letter to the editor about it. This is the paper on verification by complete testing. I'm not saying you shouldn't use test cases, I'm saying use anything you can get your hands on. What this does is zero in on where your problems are.

Now, as to your second point, it's quite true that when you write a messy program you have a hard time specifying it. In fact, the specifications tend to look about the same as the program. You have a much harder time verifying it.

R. Balzer: The theory about the number of errors predicts that when you increase the size of each procedure or each task,

you're programming beyond a certain point and you're going to have errors in it. The whole idea is to break things down into manageable pieces. The theory says that this is the best way to do it.

- D. Luckham: Great. Now we've got this nice new module construct in the language and we'd like to understand how to use it to break programs down without necessarily making them less efficient in runtime. For example, I have a benchmark program from CDC used to test their PASCAL compiler software. It computes the youngest uncle of a person from a database of people. Dave Fisher made the point last night that really the problem is not runtime speed but memory space. That's the sort of thing that would happen with an applications package from an outside source.
- J. Cross: Suppose we imagine your whole system to be implemented for the common language. And suppose you give this system a program to be verified, and the system works for a while and then comes up with all sorts of information such as "this variable cannot be accessed until it is initialized." Such information is obviously of great interest to the compiler which is trying to emit good code. Do you have any proposals about how your tools could communicate that information to the compiler?
- D. Luckham: No, I don't. We are starting to implement the language that we have now. In other words, when you give a program to a verifier you go through the first ten percent or fifty percent of the compiler, namely scanner, parser, to an internal format. Now the code generator can use that just as easily as the verifier. What we have to do to go from a verification to a compiled run is to write the code generator. That's what we're going to do. Then we're going to try to research the issue of how we might alternate between code and influence the code generated by the code generator. Right now I have just not thought about that. My students may in fact be further ahead on that than I am.
- S. Gerhart: It seems like it might be time to forget the notion of a compiler, or to revise the notion of a compiler. ... There are lots of things you can do with programs that can be considered verification, optimization and so forth. It might be worthwhile sometime in the future, in the context of this environment, to go through a series of tools, see where they overlap, see where they might be partitioned up into a finer set of tools that when combined back give the components we're used to seeing like verifiers and compilers, but, in fact, overall achieve a much greater effect. Someone brought this up yesterday; the front end of a common language should be shared among various tools. The point I want to make is, simply, there's a lot of fuzziness in the notion of the compiler and the verifier, and we ought to think about what the terms might be to be revised to be.
- D. Luckham: I tried in a recent paper to talk about the runtime error checking version as sort of a compiletime

verification. My coauthors forced me to take it out because compilers don't normally do this and it would confuse the readers as to what I meant by compiletime verification. I agree with you, I tried to do it.

- S. Gerhart: Transformation is a good example of something that would be highly useful in a context of verification. You can prove a program correct and then transform it, preserving the correctness, to another correct program. That might be the sort of thing that fits into a compiler or a separate component. It's very fuzzy.
- D. Luckham: On that point, I don't wish to adhere to this particular verifier design as something that's going to persist in the future. I think starting from very high level specifications and gradually transforming them into code that some system understands might be a fruitful line of attack.
- R. Glass : I'm a little disturbed. I see an enormous dichotomy between what we've talked about so far and my perception of verification as it currently exists in the greater computer community. My characterization of what currently exists is that we tend to be using the worst of the late 1950's technology to verify a computer system. We're so far from the technology that we've discussed here that I'm not sure it's achievable from where we stand without combining some intermediate steps. I guess I'd like to challenge this group to help find those steps.
- J. Bladen: I'd like to add to that with a question. Are you saying that you have the technology at the present time to prove a scanner/parser version of DOD1 where we could say that this root compiler is a proven piece of software and we can predict its characteristics? Is that available within the scope of DOD1?
- D. Luckham: What we have done is verify some standard properties of the scanner and parser for a PASCAL compiler. Now, what that would mean would be that it should not be too difficult to do the same thing for the scanner and parser for DOD1.
- J. Bladen: One of the hangups the Air Force has on a standard compiler is inability of proving the compiler itself. If we can get some sort of indication that we can prove a compiler, then there's a strong possibility of changing the whole way of doing business within the Air Force.
- S. Crocker: I don't understand. You're saying that the Air Force doesn't want to use higher-order languages because they can't verify the compilers with them?
- [No reply.]
- S. Di Nitto: One criticism that is made of these higher order languages is that people cannot get as close to the machine as they could with assembly language. ... Not only is your

program logic a possible source of error, but your compiler can do something to screw up the language along the way. There's this mistrust of the compiler. The point Jim Bladen was trying to make is that it would be great if we could have something that would certify that this compiler produces 100% correct code in every case.

- S. Crocker: This strikes me as very, very strange. Independent of any verification technology, a lot of the world has experience with using compilers that were written by reputable organizations. I'm sure there are compilers with bugs in them, but it's a lot easier to debug the compiler by brute force over a long period of time and trust it, than it is to debug every instance of assembly language code.

- S. Di Nitto: Over a long period of time -- nobody wants to be the first to do that. Why should your project be held up while the compiler is being verified?

- S. Crocker: You're doing the opposite; you're repeating first experiences time after time.

- S. Di Nitto: An unverified compiler is going to cost the Air Force a suit, because if we hand a government furnished compiler to a contractor and he says "oh, look your compiler just errored, and I'm going to have to charge you another six million dollars" your whole project is over cost.

- S. Crocker: The problems the Air Force has clearly have nothing to do with technology, nothing to do with verification or languages, but only to do with management.

- S. Di Nitto: Exactly. The technology is there, the management is not.

- V. Schneider: It was said earlier that DOD funds about half the software work in the country, and it behooves us to address these problems also. There is a problem when you get a contractor doing a one or two million dollar project, and that contractor also gets a new compiler that hasn't been wrung out. And the compiler not being wrung out contributes to a six month slippage of that project, the software may be one or two million but the rest of the project is another ten million. That's a lot of money.

- D. Luckham: Sure, but no reason not to wring it out. [Other agreement.] You need a technology for wringing it out much faster, and for being much more certain about it earlier in the game.

- S. Di Nitto: Of late there has been a lot of interest in using higher order languages. There are a lot of languages, the C language for example, that have been around for a while. People have used it. It's not on the standard list, but we've tried to talk people into developing a compiler for C on the 11/45. They say, "oh no, a brand new language, we'll go with one that's been proven out." It's a new type of battle. Quite frankly, it isn't a management problem, it's

a problem with the bloody contractors where the compiler becomes a scapegoat. When it comes to a court of law, who understands compilers? The contractors can say something like "the government gave us this thing and it screwed us all up."

[Unknown]: Just to understand the magnitude of the problem here: we are using two of the most reputable compiler builders in the field in our project. The Jovial J3 compiler turned 1000 errors in the first three years of use. In our B1 project, the J3B compiler turned 500 errors in the first two years of use. We believe those are typical figures for today's technology on the well-verified compilers. Those compilers have both been through the validation process as defined by the Air Force and passed, prior to the turning of those errors. So there is a technology problem in the verification.

S. Crocker: Even if one believes all of that, what are those figures in contrast to the enormous number of errors that you have to be inserting one at a time, by hand, into the assembly language?

[Unknown]: I'm not arguing that we should reject higher-order languages to go to assembly language. I'm just trying to say that there are problems with the present state-of-the-art. In the Air Force you have to sell everything you do, and right now we cannot sell the concept of a proven compiler.

D. Luckham: Just to answer your original question: the answer is no, I do not have a new technology for compilers right now. We're working on it and I would be hopeful of a reasonable approach to verifying compilers in the next year or two. But they might not be DOD1 compilers because I think there are certain constructs in DOD1 that compiler writers themselves are going to have a time with.

W. Teitelman: I want to make two points. One, where you said that your programmers had a great deal of confidence in assembly language because they could see the actual octal instruction that was produced. Implicit in that is somehow the idea that each instruction is going to execute correctly every time. For example, that there isn't going to be some sort of machine failure. That does happen. Or, in the case of the machines we're coming out with now, where most of the instructions are microcoded, very often you'll run into some peculiar problem which just happens to relate to some timing function or some collision on some who knows what, and the programmer that is writing the program looks at the program, and the program looks right, and he says "it works, I mean it should work." Then he runs off to the machine guide and says there's a hardware failure. In the same way, a person writing a program in a higher order language, when his program doesn't work, runs off and says the compiler didn't work. The probabilities in those cases are usually something like 95 or 98% of the time it's not that problem, it's higher up the line.

The question with the compilers is what sort of risk are you willing to take in terms of what the benefit is? In the Air Force case you don't say, "we're not going to release a new plane until we've proven it's not going to crash." I mean, there's a certain period of testing beyond which you say the advantages seem to outweigh the disadvantages, or we're willing to accept this risk, given that there is a certain level of certainty. Then you go with it. You have to take the same pragmatic approach with compilers. If you wait until you can say that they are absolutely 100% verifiable, you're giving up a lot of leverage right now.

J. Cooper: You missed the point of enforcing a compiler on a contractor. Even if there's only one bug left in the compiler after, in the case of the Navy, nine years of use, if it turns up on a Boeing contract, Boeing can then say, "Hey, because of that bug it cost me six months and three million dollars."

W. Teitelman : I also want to say one more thing. In spite of the error rate that we found on our compilers, we also felt that the experience of using higher-order languages on those projects was totally successful. We succeeded in spite of those error rates.

On the contractual question, we contracted with those compilers ourselves so there's no turning face on the government for a faulty DOD compiler.

S. Crocker : Plus, the verification of the compiler isn't going to solve the general problem that the contractor has a claim against the compiler if the compiler produces code that is too slow or takes up too much room, or the compiler takes too long and burns up too much machine time compiling. These are problems that are not directly addressed by any verification technology that is being developed. Such issues also open the door for a contractor to make somewhat unrelated claims. It sounds like a very good game for a contractor and a sort of lack of confidence in the game on the Air Force side.

J. Bladen : I think I can safely say that if we had a proven, verified compiler for the DOD language, that the Air Force would consider going to a standard compiler. This is based on inputs that I've had from people within the Air Force. Right now, the reason we haven't is that the technology does not exist.

D. Luckham: Let me be quite clear about this: the technology is not here yet. It's a research area. Fairly simple compilers may be possible in say one to two years. For a compiler of the difficulty of DOD1 it may be five years before you can prove substantial properties of it.

R. Balzer : I've sat through similar sessions before on the NSW projects. There again the whole reason for the project was to inject new technology into the existing DOD environment. A lot of the issues we faced were technical but the

overriding ones were managerial in nature. Steve happened to be the ringleader at that point in time. He spent a lot of time dealing with procedural, procurement, and contractual issues relating to the way DOD went about its software business. I think there's an awful lot of opportunity there to change things, given that we can provide some technology. I think the issues are outside the realm of our set of competence in this group to deal with. Maybe there ought to be another set of people convened to talk about how, if certain technological advances are made, the military ought to restructure the way it does software and hardware computing business. I don't see the advantage of continuing this line of discussion with this group of people.

- S. Gerhart: From my standpoint it's not proving alone, and it's not testing alone, but it's some combination of those two, combined in terms of testing some components of the compiler, proving some components, rather than an extreme amount of testing or proving. There are all sorts of combinations of these two notions which I think would give a much higher degree of confidence, a much more effective technology, than either the old ways of testing and the new, not yet always achievable, ways of proving. It's absurd to split these two completely from each other, when, in fact, there might be a very effective combination.

[During the course of some scattered informal discussion, a brief mention is made about the verification of microcodes. No really substantive issues are discussed.]

- P. Wegner: I was rather impressed with the remark that although the Jovial compiler had errors, they were able to successfully complete the project. I think that maybe the DOD people are frightened by errors in compilers. Accordingly, maybe we should instead of just living with compilers that have errors in them, we should increase our level of confidence in being able to handle errors. Now, in law there's a maxim that the amount of force used committing a felony is a dimension of the threat. Similarly, when there's an error, the amount of effort in handling the error should be commiserate with the error, and this should be written into a contract in some way. We should increase our understanding of how to handle errors, and here again we need to understand the mechanisms for maintenance and handling of errors. Probably the horror stories are not due to errors in the compilers but due to the management being unable to deal with errors. We should tighten up on it.

- D. Luckham: It is possible to contractually specify the degree of effort which you apply to particular errors by defining something called the priority of the error and assigning a schedule of requirements which satisfy the repair of that error.
- S. Crocker : What is your favorite list of problem areas in DOD1 that are likely to be difficult to verify (or compile)?
- D. Luckham: Modules, dynamic invocation of processes, exception

handling, for starters. In the case of the Green language there is the CONNECT statement, not to mention generic routines, parameters to processes, among others.

R. Balzer: In other words, you could handle assignment statements!

Session 4A: Technology for Compiler Validation
Victor Schneider, Chair

- V. Schneider: What I propose to do is bring up the section of PERBLEMAN that has to do with compiler validation [Section 5.2], for this is my one chance to stand up and say what I think about the subject. I have two speakers who will present their outlook on the subject, Susan Gerhart and Sam DiNitto.

[Schneider goes on to summarize section 5.2 of PERBLEMAN. In addition he presents some of his personal viewpoint on compiler validation. He first observes that in order to validate a compiler it must be defined in some formal sense. His particular bias for formal definitions is towards attribute grammars and translation grammars.

Schneider further suggests that it is useful for validation purposes to define some abstract machine which is the target for the compiler. Such a machine would cover a class of computers that are expected for DOD1 application areas. He then concludes his session introduction with the following remarks regarding the use of an abstract machine to aid the validation process.]

... The thing that I'm driving at here is that ... it is possible to specify the translation so that regardless of the algorithm that you use for doing the translation, we can expect the object code sequence in the abstract machine level to be the same for different translators. So, ... one way of validating would be to feed in a series of test programs and verify the abstract machine between compilers. And then, ... showing that the mapping from the abstract machine to the target machine is correct is a way of getting at the final proof that the compiler is correct.

- R. Sites: I completely agree with that approach, but if your verification compiler says that when I compile a program I get very specific abstract machine output, why in the world would you want more than one translator?
- V. Schneider: As far as I'm concerned you could have written the translator in its own language or in some other language that is more easily available, like FORTRAN, and move it around.
- R. Glass: I think what you are saying is that every single optimization of all the processors has to be identical because otherwise you won't get identical abstract machine code.
- V. Schneider: There is some optimization applied in the semantics here, and you may even conceive of an extra pass that's standard that does abstract machine code optimization. But up to that point, you've got a standard compiler. And after that point you may have some post optimization, but that's a very specific set of semi-localized transformations on the actual object

code

R. Glass: Would you say the abstract machine code is being checked, or are you talking about the language object code.

V. Schneider: I'm talking about the object code.

P. Wegner: Are you suggesting the abstract machine should be standardized?

V. Schneider: Yes. It's a DOD1 abstract machine.

P. Wegner: In that case there seems to be a consensus in the previous meeting and in this meeting, and it might turn out to be quite important to standardize

V. Schneider: We have enough models in the past already. There is the Janus system of Bill Waite and there is the PASCAL abstract machine. There is no reason you can't talk about doing that. And once you get down to that level, you are not that far above any particular target machine that you are aiming at.

P. Wegner: You can talk about doing it but then agreeing on the specific thing

V. Schneider: I was proposing this as a model for setting up a validation process. If people have alternative technologies that will work as well without assuming an abstract machine, that is fine, I'd love to hear it.

Col. Whitaker: Two things that bother me. The abstract machine that you want to go to for different real object machines may be very different for something like a vector machine or what have you. Optimization may be very different if you have to revectorize the whole thing. Another thing, as you pointed out, if you were going to validate down to the abstract machine level and make an abstract machine, then obviously you just take the whole thing into a compiler. That is the technique that has been done. It is the technique that not everyone is happy with because some of them don't generate very good code for the strange machines.

V. Schneider: I was talking about classes of machine. I see it as a very valid thing to talk in terms of a class of microcomputers and a class of minicomputers. I realize that there are great differences between Cyber 70's and IBM 370's, for example.

R. Sites: I'm going to talk a bit tomorrow morning about existing Pascal technology. In fact I'm working on an optimizer for the abstract machine level, which is machine independent, from which we will generate good code for the Cray-1 and the LSI-11. I consider those to be reasonable extremes.

V. Schneider: The main problem in getting something done is finding a closed form. A closed form is always a simplification. You take all the rough edges and you saw

them off, and you say, "I'll take care of that tomorrow." If I can take care of the close form first I've got that much done. And I can call it a partial success anyway. The minute you start throwing special cases at me I say, "Yes, they exist" and "No, I don't want to handle them today." I also say technology of formal language specification is being strained when you try to handle every special case possible. It is really much simpler to conceive of a class of machines and a simplified syntax and attribute grammar approach.

[Susan Gerhart now begins her presentation.]

- S. Gerhart: I'd like to go back to some first principles since this morning's session on program verification, in the sense of proving, was somewhat terminated. I'll continue a little in that vein. I'd like to take the position that the best type of validation would really be some combination of testing and proving. The reason for making this claim is to try to get you to think about some combination which might be more effective than either one individually. Of course the problem is that most of us are experts in one field or the other Proving is of course newer and very tentative. Testing is older and since we are kind of divided up into two camps, the testers and the provers, I just want to raise a point that may be worthwhile to think about. [Viz.,] an effective combination of testing and proving.

For example, some combinations might be to, instead of doing an extreme amount of either one, do some moderate amount of each sort in parallel; a moderate amount of testing, and a moderate amount of proving in the usual way on all parts of the compiler or whatever program you're working on. We might also think about doing one or the other in the cases where, if we could determine this, each one is most effective, most appropriate. Or we might actually look for some sort of unusual combination of the two of them. We might think of doing testing and then based upon the results of testing continue with some sort of a proving argument. Or we might, and in fact there are some theorems that indicate that this is feasible, look for general types of proofs which tell us that if we select a certain set of test data and if we get the right results on that, then in fact the program will be correct. There might be some novel combinations here which just haven't yet been explored.

Why should we do both? Well, testing happens to have a lot of strong tools. Just to be provocative, I might say that this might be a case where tools are in fact dangerous because there's not a whole lot of theory in testing which tells you why one tool is better than another tool, or one strategy for testing is better than another. There is, in fact, a very weak (embarrassingly weak) theory behind program testing. Proving on the other hand has a strong theory but of course it's very hard to carry out, and the tools are currently quite weak. So we have this problem that testing, which is done all the time, isn't all that well understood. Whereas proving, which is fairly well

understood, is still very hard to do. But, in fact, I think if you look at them a little more you'll find that there are various ways that they really have complimentary capabilities in terms of ability to detect errors, or ability to convince someone that the validation process has been good.

... Based on an intuition and what we tried to understand about testing, there are a couple of approaches you can take. One is the black box approach, selecting data, working from the specifications but not necessarily getting into the internal structure of the program that you're testing. Another is faced in the opposite direction, working strongly with the structure of a program. For example, trying to exercise all parts of it but not necessarily being so concerned with cases that might occur or be expressed more naturally in terms of the specifications. It seems like the most effective way of testing just in a very general sense, is to work from specifications, backing that up with some sort of monitoring of the results of executing those programs, supplementing what you feel that you have missed in terms of exercising of the program with additional data.

[Gerhart discusses the classes of programs that can be used as test data for compiler validation. These include correct programs, "almost correct programs (i.e., those with subtle errors), and grossly incorrect (really bizarre) programs.]

... There are, of course, a whole range of testing tools and it certainly seems to me that to say that you've validated a compiler when, in fact, you may not have ever executed some statement of a program would be totally absurd What I'm suggesting is that the full range of these testing tools which are available for whatever language the compiler is written in should be applied to it. And finally, as sort of a certification aspect of having people really look at the test data that is selected, I think there's intuition and there's a little bit of theory which would allow you to do a pretty fair inspection of the test to judge the quality of those. And the test sets of programs might actually grow also, as for example if one compiler passes the validation test and someone later finds an error by trying to compile a certain program, that program should probably become, thereafter, part of the validation test set. Another aspect is that, of course, if a compiler's ever to be modified or to be understood, even though it may pass the above test, it should also be well enough structured that it can be maintained so that the code of the compiler should, in fact, also be read.

Turning to program proving, there's really a whole different set of issues that come up which are worth looking at both from the standpoint of proving and testing. When you try to prove something you have a major task which requires you to factor both what you want to prove and the components that you've proven about in as many different ways as you possibly can. You simply have to break down the task. You

can't talk about proving a compiler in its entirety. But another way of thinking about factoring this process is the various properties which you might want to prove about the compiler. For example, you might want to know that a set two of programs that the compiler accepts is equal to the set of legal programs in the language, according to the semantics. You would like to know that if a program is accepted the compiler will produce code for that program. This sounds sort of trivial but when you talk about various space requirements it may be a fairly complex thing to show. If a program is accepted, and of course this is the major thing that you're concerned with, how do you know that the generated program is equivalent semantically to the source program? And you might want to break this up to make it modular, the run time package being separately proved. So there are lots of different ways of factoring the properties to be validated. Again, some of these might be dealt with by proving; some of them might be dealt with by testing.

Another way of factoring this task, and I think that this is really important, that the compiler is a system of components. We're used to thinking of it as one big sort of a monolithic entity. But it uses a lot of data structure modules and those data structures are not all that peculiar to compilers; they're used all over the place. Various tables, trees, streams of code, streams of characters. You can think of a compiler in much more abstract terms than it often is. Algorithms can also be broken out, for example, for parsing, register allocation, and optimization. Breaking all of these down can make the task of proving much more feasible. For example, if we were to really try and verify a compiler, I think we would go at it by taking these components one at a time. People have already studied trees extensively as data structures; they've studied various types of transformations on programs which correspond to some of the optimization algorithms. Tables and so forth can be verified independent of compilers if they are sufficiently generalized.

[Gerhart briefly mentions some proof strategies which make use of program transformations.]

I've just tried to mention a few of the different aspects of testing and proving and I think it might be worth looking at, in terms of a long spectrum of time. some combination of these two.

- R. Young: The preliminary PEBBLEMAN document talks about compiler validation in terms of testing and you've introduced an alternative method, if you will, one of proving or some combination of testing and proving. It seems to me that everytime you change the object machine or the interface with the target machine, such as the operating system, you're going to have to go through the whole process of retesting the compiler. Do you think that we have a technology for determining the most cost optimal method of validating compilers? How do we know that a combination of testing and proving will test and validate it entirely? Do we have such a thing?

S. Gerhart: No, of course we have no real cost measures whatsoever on proving except it's almost infinitely costly. But testing is very costly also.

R. Young: So how do we decide the trade-off?

S. Gerhart: I don't know but I think that it is worth thinking about. Just take testing for example. How do you know when to stop testing? How do you know how much test data is enough? And if you don't know that, then the other sort of cost benefits of proving are also hard to factor in.

-----: I have an idealistic question. In the long run, in a program which has been proven or in a module which has been proven to be correct, what is the role of testing for such a module other than performance kind of issues? Doesn't testing sort of disappear as correctness becomes more formally achieved?

S. Gerhart: Okay, I would never go through a big proof effort on something that hadn't been tested first. Testing is a very effective way to get out the real dumb-dumb errors. But I think that proving can go much beyond [testing] in terms of confidence. Now if we had a sufficient theory in testing then that might not be true. But lacking a theory, it's very hard to draw strong conclusions. So I would think of proving as being a phase which follows testing. But you can use proving for lots of different reasons, perhaps the least of which is to produce a proof. You can use it when you need to understand something very, very well and you're finally ready to formalize it completely in terms of specifications and really trying to put together all that you know about a subject. A proof extracts all the knowledge that you have about a subject so you can use proving when you have a great deal of understanding and are really ready to get that all down once and for all. That is when it may be most important to attempt to prove--to increase understanding, to force out kinds of reasoning that you might not come up with at all during testing.

V. Schneider: We have two things here by the way. One is that the compiler is the same as the specifications of the compiler. And the other that the compiler generates correct programs. Just because the compiler is the same as the specification doesn't mean the specification is correct.

S. Gerhart: Right. So correctness ... is a matter of consistency.

J. Knight: You said there was no theory about the generation of test cases. Suppose that you were to challenge everybody at this workshop to submit one test program when the language is finally defined and ask everybody who was here to be as devious as they possibly could be and as challenging as they could be. How good would the compiler finally be if all of those programs went through it and checked out? I suspect there's enough brain power and bloody-mindedness in this room that it might well beat a very good compiler.

- S. Gerhart: I think that's right. Testing very often is done trying to confirm that the program does the right thing so you stay away from the data which might show that it was bad. In fact, testing is most successful in an adversary sort of situation where you're really trying to beat a program to death.
- S. DiNitto: I really don't agree that having everyone at this conference generate test programs would give you good results. We've seen cases where a well-used compiler over a period of ten years still had bugs. ... One machine, the 1604, was almost ten years old and we counted five errors in the syntax analysis package. Now you would have figured that that compiler would be correct in at least the syntax analysis portion. The same compiler was found to have bugs in it during training courses.
- S. Gerhart: Well, one of the problems often is that people don't know what the compiler is supposed to produce. So those errors may have in fact been recurring over the years, but without sufficient specification or sufficient clarity as to exactly what was supposed to happen, they may not have been recognized as errors until this point. Is that possible?
- S. DiNitto: I assume it is, yes. We experienced this in a case. An error can be there for quite a long time. And I think it's all the more important that we get it out.
- S. Gerhart: Well, let me raise another point. Maybe you can't expect to ever have a perfect compiler. What is the level of tolerance which you can accept and what are the measures that you might bring in to allow you to live with just a few bugs?
- S. DiNitto: We set a standard in the Air Force for let's say K-3. It has to pass our set of validation tests ... 100%. We used the compiler which passed the same tests and there's been in the neighborhood of 50-70 bugs. ... I don't know, is 50-70 good or bad?
- V. Schneider: 50-70 during the development process, during the validation process, after the final week?
- S. DiNitto: After we accepted the compiler.

Session 5A: Compile Time Tools
Martin Wolfe, Chair

M. Wolfe: I thought of all the tools used at compile time, and I came to the realization that obviously the compiler happens to be one of the most important. So what I would like to do is discuss what we expect from a compiler and what are the implications of these expectations of those requirements on the structure of a compiler.

The first thing you could look at is what a compiler provides; what are its outputs. Of course a compiler can produce object code. Rather than call it object code, it might be better to call it code for a virtual target machine. The reason for this is we might have trouble separating that part of a compiler which is runtime support for a bear machine versus that part which generates what might be called "actual" machine code. For example, for machines with a sophisticated operating system, much of the runtime support will be part of that operating system rather than part of the compiler itself.

The compiler should also produce documentation aids. What are documentation aids? Object code listings, for example, source code listings, error messages, symbol tables. But what do we mean by these and what format should they be in? If you look at object code listings, should they have a correlation to the source listings? In source code listings, perhaps statement numbers should be indicated as well as variable scoping, typing, etc.

Whether we standardize error messages or not can have a tremendous impact. If we say we will have to have standard error messages then we may have impinged on the parsing technique one uses. For example, we might have to detect errors at different times, or if you use one parsing technique, an error might not be detected at all. Perhaps we should set guidelines to the implementer of what should be the format of error messages. For example, you might want to say that error messages should be in a language that relates to the source code. It shouldn't be that you have to look in a table to find out what the error is. Now that might have impact on the size of the machine that you want to host the compiler on.

There is a whole range of tools that can provide statistical information. Should they be part of the compiler or should they be separate tools? A compiler might provide information on the the amount of resources consumed at compile time. Other issues include: should we provide a restructuring tool that reformats the listing in some way, indicates program control flow, data flow, etc? Should the compiler do full type checking and/or interface checking? Should a syntax-checking text editor be part of the compiler or should it be a separate tool?

These are some of the issues that I'd like to discuss this

morning. To start off, I have two speakers. The first is Captain Bladen who will be followed by Dick Sites.

- J. Bladen: Unfortunately my briefing on the way we planned to do this within the Armament Lab is back on my desk at home. Maybe that is where it belongs. We at the Air Force Armament Lab have taken the position that the best way to do business with the small computers used in missiles is to have a standard retargeting compiler. My definition of retargeting is that you have a parser and scanner combination which makes up the compiler itself. You then have an intermediate language which inputs to a code generator. The code generator is a swap-out item so that if you write the source code program in JOVIAL and you want it to run on the INTEL 8080, you load in the code generator only for the 8080 and target your program to the 8080. You may then change vendors to some magic computer, and instead of going back and regenerating the program, it's done now. Instead of writing new software, we will simply swap in a code generator for the magic computer. There are a lot of implications in this statement and one is that we are going to have a standard compiler. If any changes come to the language we'll make those changes to the one compiler; it will be automatically reflected throughout all code generators, all implementations.

[Bladen's work in this area is discussed further in his position paper, as well as in other conference sessions.]

- J. Knight: What you describe is precisely how the MUST program is organizing its compiler. We have an Avionic programming system which is organized with a standardized front end with multiple code generators for various machines. The various validation and verification tools operate on the same form of the intermediate language as do the code generators. The Huston Space Center has got an IBM version of exactly the same program written in XPL, and the front end is identical. They have code generators for various machines. The reflections that we have so far are that in an Avionics system, at least at our end, the whole thing seems to be a very satisfactory arrangement. There was a lot of debate yesterday about the use of the front end of the compiler for V & V tools; in our experience anyway, it is a good idea.
- C. McGowan: As with DOD1, there was an evaluation of existing languages and it turned out that for their purposes they decided it was probably better to design a new language rather than build on an old one. To standardize on an intermediate language it might also require a similar effort.
- J. Bladen: And it should be an object of this meeting that we make that recommendation.
- C. McGowan: This can have implications on a standard DOD1 compiler. If you advocate a single source for the various compilers that would be distributed throughout the field, the question of field modification, and in fact the whole

issue of configuration control, becomes a significant problem.

So, for example, if we had a cross compiler that ran on a large machine, and compiled code for the 8080 instruction set and then we did a modification to it, then that modification would have to be sent to all the installations that have the cross compiler.

- J. Bladen: If that modification was accepted as part of the base compiler, the central agency would be the one that would do that. That is the way I see it. There will be only one version of the compiler at any one time. This is done in the Air Force now in the finance department. If a change is required by someone out in the field they request a change be made by a central agency. This agency decides whether or not it is a valid requirement. They make the change. They send out the tapes and on a certain day at a certain time, the system is brought up so everyone across the whole country has the same system at the same time.
- K. Bowles: What language is your standard front end written in if you have any concern about portability of that front end?
- J. Bladen: The front end is written in JOVIAL and this provides portability.
- D. Loveman: What is the intermediate language? Are you referring to just the internal representation of the statement sequence of programs or are you including other information such as the symbol table and flow graphs? The idea of a standard intermediate language sounds very desirable. I have found in practice, however, that the design of an intermediate language is very often dependent on the desired performance characteristics of the compiler. For example, the decision whether to imbed information in the intermediate language, keep it in an additional table for reference, or recalculate it each time it is needed, clearly depends on the required space-time characteristics of the compiler. I wonder whether or not the technology exists to pick a single standard intermediate language that is going to be usable over the whole set of potential DOD1 compilers.
- J. Bladen: We are going to have a group to study this particular problem.
- R. Glass: I'd like to echo this concern with the knowledge of the internals of the existing JOVIAL J73 compiler. First of all, the compiler is an excellent design; nevertheless the intermediate language had a design goal of being totally independent of the symbol table, and in fact that design goal failed. It's also true that for some code generators there are machine-dependent characteristics which still require front end changes. What I'm trying to say is that there is concern about the level of our understanding of good intermediate languages and proper techniques for using them. It is probably premature to standardize here. We are

getting close, and I think we have learned a lot. I think that the J73 compiler is an excellent example of the directions we have to be pointing toward, but I don't think we know enough to standardize.

- J. Bladen: First, I want to answer the first thing you said and that is that, yes, the symbol table is part of the intermediate language. That's something that was an oversight on my part. There are things besides intermediate language which should be kept around in order to deparse to bring back the source code, but these are implementation issues that I'm not really addressing at the moment. What I really would like to address is whether or not the technology is there for a standard intermediate language.

As far as your statement of whether or not the best intermediate language is now available, I don't see how that's possible. But the best DOD1 language is not available. However, there are sufficient intermediate languages. We should choose the best of the sufficient intermediate languages and say this is the one we are going to use. Either that or there is going to be a proliferation of intermediate languages just like there is a proliferation of FORTRAN languages.

- D. Loveman: My experience has been that the design of an intermediate language is a function of three things: the source language for which the compiler is being constructed; the performance constraints imposed on the compiler; and the target machine architecture. We will have a standard language, but do we have a standard set of constraints on the compiler and do we have a standard machine architecture? I don't think so. Even if you pick a standard representation form like a tree structure or QUADs, you know what the intermediate language is going to look like but the details of its implementation will depend heavily on items other than the language.

- R. Glass: I'd just like to throw out one other thing related to intermediate languages. And that is, the possibility that a standardized intermediate language may result in hardware which executes that IL, and makes trivial the tasks of writing code generators off that IL and/or interpreters for that IL. I think there is a lot of down stream fallout from standardizing on IL. That ought to be taken into consideration. Once the standard is established, a lot of stuff gets frozen and locked into that standard.

- C. McGowan: Code generation is one aspect of program representation with which we are concerned. Mention was made of doing editing, and editing not just as on source text, but on something that reflected the program structure. So the kinds of compiletime operations that we want to do on an internal representation of a program besides code generation would have an impact on what would be that internal representation.

[R. Sites now begins his presentation.]

R. Sites: I am going to talk a bit about implementation for PASCAL and I think that may settle some of the questions in your minds and quite likely will trigger more questions. This is the general structure of portable PASCAL compilers stemming from the P2 portable compiler of Wirth, et al. [Sites refers to a prepared viewgraph.] They wrote the PASCAL compiler in PASCAL in about four or five thousands lines of PASCAL code that compiles from source to an intermediate pseudo-code. Then there are separate translators that take the pseudo-code and generate code for specific machines. Existing things that have been running in the field for over a year are code generators for the IBM 370, the CDC 6600, the CRAY-1, the PDP-10, and the UNIVAC-1100 series. It's a very wide spread single compiler base of this sort that is filtering to the University system, and to some extent the commercial world.

Another possibility, once you compile to an intermediate pseudo-code, is not to compile that to machine language for a particular machine but to interpret the pseudo-code directly. That's an approach that Ken Bowles used in UCSD PASCAL by writing a small interpreter for the DEC LSI-11 that will interpret pseudo-code directly. Rewriting the interpreter for another micro such as the 8080 (which they also have done and have had running for quite a while) that interprets the bit-for-bit identical pseudo-code, can be up and running on an 8080 in a few months.

[Sites proceeds to give an implementation-level slide presentation of the UCSD PASCAL system. Details are omitted here.]

- P. Eastwood: You emphasize that choosing a very simple assembly language stack machine may make the compiler portion easy. Did you find later that that made the code generators harder?
- R. Sites: No. Any reasonable code generator is going to take a bunch of temporary names and throw out some of them and remap the rest into general registers, index registers, accumulators, etc. I feel that the choice of forms here between stack machine or accumulator machine or 3-address register machine, is really a red herring discussion. Whatever form you pick, it's not going to be perfect for everybody. But if you do a decent job, it's going to be good enough for everybody. That's the best you can do.
- K. Bowles: In our implementation on the microcomputers, we are not going through assembly language, we are interpreting a compressed P-code and the whole things hangs entirely on the idea, if possible, to express that P-code in a form which can be depressed efficiently so that one can get a single pass compiler on to the microcomputer. Our experience with these micros and various others' experience suggests that we don't yet know enough about the process of expressing that P-code to think that it would be possible to standardize it.

- R. Sites: I agree with that. I will later take the position that DOD1 should not in the first two years standardize the P-code. It should rather take the position that as the problem is understood, one of the goals is to standardize on a P-code that everyone is happy with.
- R. Glass: Doesn't this approach to an assembler like P-code make it difficult to do global optimization?
- R. Sites: I think I would like to have people stop asking questions for about 10 minutes. I want to give you a very specific instance of P-code. It is the assignment statement $A[I] \leftarrow B+C$.

[Sites proceeds with a very detailed discussion of the P-code generated by this assignment statement. The technical details presented are irrelevant to the general level of discussion which follows.]

- [Unknown]: Doesn't your design imply that your compiler, which is common among all machines, has built-in logic that decides what variable is going to go into what type class. You have to assume that there's going to be a memory hierarchy, and enough compiler logic to decide what goes where.
- R. Sites: That's true. In order to do a decent job for a class of machines, the front end assumes that there is a memory hierarchy and has a small constant describing how big each of the hierarchies is, the order of the hierarchy, and essentially a small cross reference table that says integers can go into these places, reals can go into these places, addresses can go into these places, etc. It's not a hard thing to capture and it's not a hard thing to change.
- C. Taylor: What prevents you from generating a new machine architecture and having the same problem? That is, having to have the compiler understand this new machine, etc.
- R. Sites: There is no answer to that argument. Whatever compiler design anyone in this room presents, you can find yourself a machine that that design would generate poor code for. Necessarily, as new machines are invented which are radically different from old machines, there will have to be changes all through this. What I am arguing for is the decomposition, which is in fact reasonable and tracks at least an existing set of machines and ones that are on the horizon for the next five years.
- C. Taylor: That applies to changes to the compiler. The compiler has to understand what such changes mean to the computer architecture for the next five years.
- R. Sites: That's not completely true. You can take out the whole thing about the memory hierarchy so long as you are willing to put up the code which only uses main memory on a machine that does have a hierarchy.

- C. Taylor: It seems like you're picking particular data structures, and you are forcing those data structures on the machine, while those data structures may not be the best for the particular machine.
- R. Sites: But it can't be the best for a particular machine. All I'm arguing is that they are reasonable enough and that so long as you don't lose any information, you can have a code generator which rips all that back out and puts something else back in. I'm going to talk about remapping storage in a minute.
- P. Eastwood: We have a standard compiler which uses parameters for different classes of machines. We felt that we didn't have as many copies of the compiler in the field as we did code generators, and that it was maintained better.
- D. Loveman: I think the objective here is not to describe the best possible architecture for the ultimate compiler for a given machine, but, rather, to propose an architecture for a good compiler for a variety of different machines. And the question I would like to ask concerns the role of, or perhaps the conflicts between, the idea of a single standard good compiler for a variety of different machines, versus an outstanding compiler for a particular machine. Do you want to have the ultimate compiler for each machine you are going to have? Just how important is optimization for this language on a particular machine?
- R. Sites: The idea of having anything even vaguely related to the target is to have an abstraction that would cover many targets. You can have very narrow abstractions which cover exactly one target, or very wide abstractions which cover many. But there is some judicious tuning available. Clearly you can bypass the optimizer if you don't need it. The optimizer at this level, I believe, should have responsibility for doing global machine independent optimizations. Anything you do to reduce the amount of computation treating the pseudo-code as an assembly language for a machine, anything you do to remove an instruction from that block of code, always wins on all machines. Anything you do to take five instructions and move to a less frequently executed place, always wins on all machines.
- M. Wolfe: That's not true. I was having discussions with Dave Loveman this morning, and optimization is always dependent on the target machine. For example, even if you take out common sub expressions from a loop, what happens when I go into a vector machine? Well, in a loop you may get two instructions for what could be done on the vector machine by one instruction.
- R. Sites: I don't follow that, but I'd like to talk to you about it at the break. Your option always, of course, is not to optimize anything, if you are going to be better off that way. If you eventually find that you need to build another optimizer box that does different things. fine. What I am trying to propose is a structure which at least

works for current machines and the current PASCAL language.

- M. Wolfe: My point wasn't saying that the structure is not good; the point is an optimizer is not really independent of the target machine.
- R. Sites: The place I'm coming from is I'm willing to not do some optimization which will be applicable to one particular machine only. I'm willing to generate code which is quite reliable but a little bit slower than it possibly could be for the alternate machines. And if you have some things which belong to a particular machine only, you do some optimization for that machine only at the P-code to machine code translation level. If you've got too wide a variety of machines or this whole structure grows for fifteen years, it will eventually fragment too much, and you need to start over. We haven't reached that point yet.

[Sites proceeds to give some specific details of global, machine-independent optimizations that may be applied to P-code.]

.. From building this we have found that a pseudo-code is sufficient; it's not the perfect interface for building an optimizer, but it's good enough. We also found that there were a few things missing. We found that in the current definition of P-code, when you do an array subscript there is nothing that talks about the length of the array. With lengths missing, there is no way of telling if an indirect store into an array could possibly corrupt some other variable. So this is still an evolutionary process of discovering exactly what information you need and what constitutes perfect knowledge. But it has been a surprisingly good base.

The final code that we get out will be noticeably better than the unoptimized code, but it will never match the perfect possible code that you get from a heavily intertwined optimizer in assembly language for some specific machine. However, one of the things we have to look at is that software design cycles are fairly long now, perhaps five years, while some hardware lifetimes, or at least design cycles of new products coming out, are down to the two to three year range. So it is not clear to me that you ever want to build a really heavy optimizer for a particular machine, because that machine architecture's lifetime may be shorter than the design time to write the compiler.

- R. Glass: I have to take serious issue with that position in the DoD computer environment. Our experience at Boeing Aerospace is that we have to stuff a lot of code into too little machine over and over again. A high quality optimizer is an absolute requirement.
- P. Sites: I agree with that. In that environment the lifetime changes of a particular architecture are much slower, and I'd like to turn that around. Why are they so slow? Is it because technology in this country can't build new aerospace

computers? Or is it because software is so locked in that you can't afford to use any other machine?

- R. Glass: It is primarily the latter.
- R. Sites: Things will never change as quickly as they could, but to the extent that you can move to new machines as they become obviously attractive, I expect those cycles to get a bit shorter. I've moved programs from the LSI-11 to the CRAY-1 written in PASCAL using this compiler system. It really works. Ken Bowles has a PASCAL system interpreter running on the LSI-11 and an interpreter for the 8080. The compilers for the two machines are essentially identical.
- R. Glass: I know a guy--a top-quality guy--who wanted to rehost a PASCAL compiler from one version of a PDP-11 to another. It took six months to do it. We all have stories that bad.
- R. Sites: Advantages of all this for DOD? First, you have to build an initial version of DOD1. You build a very simple parser that doesn't try to do anything fancy at all, since you have a language that is still stabilizing. Then you build brutally simple code generators. You build no optimizer to get off the ground. You can separately verify or prove or certify the front end and various back ends. Certify more than one front end if you need to; verify that they generate the identical P-code sequence or logically put out equivalent ones. When you go to the new target machine you are looking at half the work of writing a brand new compiler.
- C. McGowan: If one were to use this approach for DOD1, there are features such as being able to synchronize on a real time clock, or schedule concurrent processes, that might influence this P-code.
- R. Sites: I would expect the things you named would have a very small influence, that there would be a couple operators or perhaps more standard subroutine calls for reading the real time clock or starting a task. I would not expect that they would in fact affect variables allocated within a particular module.
- C. McGowan: Take for example concurrent PASCAL; what changes would you have to make?
- R. Sites: Other than a standard subroutine call, to start up the new task, none. I take that back. There's one other thing having to do with side-effects. In both that environment and in the environment where you have a language in which you can say, "this variable matches this machine register," you need to be able to tell an optimizer "this variable could change at arbitrary times and, hence, don't use it in common subexpressions." That's the one piece of information I can think of that would be needed in addition in a multi-task environment.
- D. Loveman: Does the CRAY-1 code-generator generate vector

instructions?

- R. Sites: Currently no. That's the second year of my research proposal. And it may turn out that the P-code is not the best representation, but if it's a sufficient representation, I'll be happy.
- D. Loveman: It seems that in the design of this compiler, one of the main criteria was exceedingly clean interfaces between the various pieces of the compiler. I think you pay something for that. One of the things you pay is the necessity of gathering certain types of information at multiple different places within the compiler. It seems, for example, that you need certain types of flow information in the compiler to help you do memory allocation.
- R. Sites: This particular compiler does no flow analysis whatsoever. The compiler takes declarations as they come in and allocates storage. It takes the statements as they come in and it generates code. It's very simple.
- D. Loveman: But if you want to do cunning memory allocation, you need flow information.
- R. Sites: If you want to do cunning memory allocation, you probably should not do it there. The thing I was speaking of before is that you could rearrange the allocation later. So long as you have the information about what the objects are and which ones must be related to each other and which ones are independent objects, then you can resort them so that, for example, the first 16 are more heavily used.
- D. Loveman: Okay. But you need flow information at least for the optimizer and for codegeneration.
- R. Sites: You need flow information for the optimizer and my contention here is that the codegenerator should do no flow analysis. Flow analysis is a very difficult thing which traditionally is messed up and I feel to have a reliable system, you need to do it exactly in one place.
- D. Loveman: But then how do you do sophisticated register allocation without flow analysis? Then that's not a machine dependent optimization.
- R. Sites: In the output of the optimizer we currently have running, when it does things like allocate common sub expressions or move an expression out of a loop, what it does is it generates assignments to and fetches from a series of names which are intended to be mapped into machine registers if you have any. And those names are ordered so that name 0 is used very heavily, name 1 less heavily etc. That is by no means a perfect register allocation algorithm. But it's fairly decent, and it maps into the huge variety of target machines. If you have two registers available, you put temp 0 and temp 1 into the registers and it does a decent job. If you have 16 available, you put the temp 0 through temp 15 into registers.

- D. Loveman: How do you minimize the number of registers?
- R. Sites: That's the job of the optimizer because the optimizer's the only place that has the flow analysis information to do the mapping correctly.
- D. Loveman: The point that I'm making here is that there's a trade-off between having exceedingly clean interfaces between the different parts of the compiler, and having a well-engineered compiling machine with systematic information being passed through in a variety of different cunning ways. And I think this [former] approach is a very interesting approach to think of as a first implementation for a good compiler, but I don't think this is a very good approach for an ultimate compiler.

I want to get back to the question I sort of got at before which is the role of an ultimate optimizer for a language like DOD1. Perhaps we're raising questions about program transportability -- how important is program transportability? You can argue the transportability of tools written in DOD1 may well be quite important. But for the programs that are in fact written in DOD1 for embedded systems, how important is it when these programs must be highly optimized? Is it not liable to be the case that you will know at some point in time that you are in fact compiling for this computer which is going to be in the nosecone of a missile for the next ten years? That guidance program is never going to change and in fact what you may want is to have the best possible optimization you possibly can applied to that program to help make it fit into that particular environment.

Another somewhat related point is the idea of the ultimate optimizer. The technology that exists for doing analysis and optimization on programs is currently well beyond what is in fact implemented in production optimizers. The main reason is the trade-offs on just how much time you want to spend in the compiler doing optimization versus the benefit you're going to get from doing an optimization. Embedded computer applications may just be the case where you're willing to say that one version of a compiler you want to have is the one where compiletime is completely irrelevant and what you want out is the best possible code you can get in the state of the art.

- R. Sites: I might read you the DOD requirements - they're not quite that extreme. If those are your requirements, you should be writing in assembly language perhaps.

[Scattered disagreements.]

- K. Bowles: You talk about optimization as if it was one animal and yet, like in this missile example, there are at least 2 different optimizations you might wish to apply to different parts of your code. Is it space or is it time? Or do you want to be able to partition your code and optimize for one purpose in one area and for the other purpose in a different

area, and so on?

D. Loveman: Green [the language] lets you say that a procedure may or may not be compiled in-line. The way that it's phrased in Green, you have to commit yourself when you write the procedure as to whether or not it goes in-line, and that's a commitment for all calls of that procedure. And based on what you're saying, you'd probably want to have a facility which says for this call, which happens to be inside four doubly nested for loops, you want to expand in-line. And every place else you don't, because you want to save space. So there's a trade-off of language facilities for talking to the optimizer.

R. Sites: The optimizer built in CRAY-1 in fact will merge procedures in-line under appropriate circumstances.

D. Loveman: Can the programmer hint at what those circumstances are?

R. Sites: Currently no because PASCAL doesn't allow any of that.

D. Loveman: I agree very strongly with the idea that runtime information about the real honest to goodness program should be fed back into the compiler. I can think of two ways of doing it, neither of which is particularly attractive. One is, keeping in mind the idea of the program development system, what you want to have is the result of this analysis fed back to the program development system and catalogued with the program so that the information is available automatically the next time the program is compiled. There are obviously certain problems involved with that. The other is by means of programmer introduced statements into the procedure giving some of this information. That has always been a problem. So you find yourself either stuck with a very elaborate system or the requirement that the procedure be modified by the inclusion of the information.

R. Sites: No, you don't have to be elaborate, there's middle ground. You can supply a very simple tool which generates statement counts. The tool that generated those counts I wrote in about a week for the CRAY-1 PASCAL compiler. Once you capture those counts on a file as ASCII characters you don't have to build elaborate tools to catalogue them and save them and associate them with a particular module and a particular date. All you have to do is have whatever optimizer or whoever's going to use them be able to read a file, and then for your starting place you have programmers say this is the file that has the counts. As you build more elaborate tools that do some of that bookkeeping for you, fine. But don't start off with the whole thing so top-heavy that it crashes under its own weight before you do anything useful at all. Build the simple, straight-forward tools, and use them a bit.

J. Bladen: I'd like to address a statement you made about the requirements within the environment that this system will be

used in. As long as it's used in missiles, it's an absolute requirement that it be retargetable. That's the most important thing. As long as we are buying competitively, which is hopefully the American way, we're going to have to be able to pull out one guy's computer and put another computer into the system.

- E. Nelson: There's another reason for having portability of the software in that as you get into more and more of the system, there are going to be common parts that may be used in a different system, and it may reduce the software quite a bit if you can reuse these parts in another system.

[The line of discussion now moves towards the topic of separation of tools versus inclusion of many capabilities into a single "compiler". R. Taylor makes the following statement which seems to meet with more or less unannamous approval.]

- C. Taylor: I'd just like to react a little to Susan Gerhart's comment of yesterday. I liked her idea that the common notion of compiler is not very useful. Rather, we should have parts which fit together very well, and if you wish to use them together then that's fine and good; just don't build me a monolith, and don't require a monolith.

[The topic of discussion now switches to the desirability of standardizing source code formats, both those output by the compiler in the listings and those used by the programmers. The topic of how statement numbers should relate to source statements is then bantered about at some length. Crocker makes the following well accepted remark to conclude this latter subdiscussion.]

- S. Crocker: I've been fortunate to be programming in Interlisp for about four years, and I hope to never see another line number!

[The discussion moves now to the area of standardized error message formats. Again, Crocker supplies some well-needed wisdom to aid the generally disorganized discussion.]

- S. Crocker: It's a question of specific cases. It is clear that you can't tell how much storage is required [for error messages] until you look at what the object machine is going to be. So running out of storage, for example, is a kind of thing that is not going to be detectable at parse time. But I think the idea of having a large class of syntax and semantic errors caught in a standard way with standard diagnostics is an excellent improvement on the current state of the art. Let me frame it as a motion. I hereby move that as part of the definition of the language there be standardized diagnostics.

- D. Loveman: Standardize all user interfaces with the language? Forms of listings, forms of outputs, etc?

- S. Crocker: Yes, in fact I have the following philosophy. There are people who like to put things on different lines and there are people who like to control their own level of indentation, and do their own prettyprinting by hand. In terms of sharing code among people, I believe it is a positive benefit to have the system decide what the format is, i.e., to have a standard system of prettyprinting. (This moves into a different area, but as long as we're talking about user interfaces.) People who are very set in their ways about how much indentation they are going to have and where they are going to put the parenthesis, and all that sort of thing, actually do a slight disservice in terms of making it more difficult to communicate code to other people. They also spend a lot of time doing that which is unnecessary. So in terms of moving in that direction, I would say yes.
- D. Loveman: After the great philosophical debate about how terrible his style is and about how wonderful my style is, I would rather the two of us program in the same style whatever it is.
- S. Crocker: You can't argue with me; I want your style. I'd much prefer to adopt your style so that you and I can read each other's code. I have less concern about where these things go. In fact, as I find that I have the opportunity to reason it out, I spend an enormous amount of time playing with it and deciding on it. In fact, it doesn't make much difference. What matters is that I can accommodate quickly to a standard set of specs.

[Scattered disorganized discussion continues for a brief period after which Wolfe closes the session with a short summary which reiterates some of the major topics discussed.]

Session 6A: Supporting A Flourishing Language Culture
Peter Wegner, Chair

[P. Wegner has summarized his opening remarks in the five page report which follows immediately.]

Language Design and Evaluation Studies

Technical Note 7: Supporting a Flourishing Language Culture (Chairman's remarks, DOD Higher-Order Language Environment Workshop, June 1978; edited version, October 1978)

The basic idea behind this session is that a programming language provides a basis for the development of a literature of programs and a culture associated with its community of users. This happens with natural languages over a period of many generations, and with programming languages over the space of a few years. We want to look at the mechanisms involved in creating such a culture and consider how the process of introducing a new programming language such as DOD/I might be helped by understanding of these mechanisms. We can gain some insight into these mechanisms by examining existing language cultures for Fortran, Cobol, Algol, Lisp, Basic, APL, etc., and attempting to understand the factors which govern the success of these languages. Understanding of the "cultural dynamics" of existing languages should help in the formulation of policies, incentive schemes, technology transfer mechanisms, etc. to encourage the development of a new language culture.

The factors which govern the success of a language are technological, sociological and ecological. For the purposes of the present discussion we may assume that our new language is a technically superior and technologically feasible and desirable product. We must consider sociological factors which govern change and technology transfer in a community of language users. But perhaps the best starting point in considering factors which govern the introduction of a new language is an ecological (Darwinian) one which views the rise and fall of languages from an evolutionary standpoint tempered by modern ecology.

Early programming languages such as Fortran and Cobol filled an ecological void and were therefore able to spread very rapidly (like weeds). Later languages like PL/I and Simula which claimed to provide a better means of meeting general-purpose programming needs did not fare as well as the first languages, in part because they could not displace an already indigenous population of ecologically successful languages. This is true to an even greater extent in the natural language field, where Esperanto was never a credible alternative to existing natural languages in spite of its claims to superior logical structure. Displacement of an incumbent language by a superior product is thought to be easier in the programming language field than in the natural language field. But the magnitude of economic, sociological and cultural investment in existing programming languages is a very powerful force preserving the status quo

The success of relatively late languages such as Basic and APL may be explained in terms of the notion of an "ecological niche". Both of these languages owe their success to filling an ecological niche not already occupied by an existing successful language. The first step in the success of a language is to find an ecological niche. By occupying a niche it becomes an accepted member of the community of languages. A language with a niche, like a people with a homeland, can later branch out and muscle in on the territory of other languages. But it must have its home territory in which it develops its cultural roots before it can indulge in empire-building.

DOD/I has staked out a claim to an ecological niche in the domain of "embedded computing". This niche is currently occupied by assembly languages, CMS-2, Tacpol and the Jovial languages. Many of the current occupants are technologically backward and it is hoped that DOD/I will be a considerable technological improvement over all current incumbents. However, each of the current incumbents supports a language culture and a group of programmers and managers that believes itself to be economically dependent on the continuing of this culture. DOD/I must carve out its ecological niche by displacing current occupants, and this task will inevitably cause pain and discomfort to current occupants of the niche.

Introduction of a new organism (programming language) into an already populated environment involves a process of conquest and subjugation of existing occupants. It is well known that occupying forces are resented no matter how humane they are or how cogent and persuasive are their arguments that they represent a better way of life. We must examine mechanisms for introducing a new programming language that are not too painful for previous human occupants of the ecological terrain, and that will allow the indigenous population of programmers to identify quickly with their new masters and become productive citizens of the new society. Such mechanisms will clearly involve both the carrot and the stick. There will inevitably be an element of coercion by means of high-level directives. But there will also be incentives and reorganization of the infrastructure so that elements of the common culture which are capable of change are allowed to take the initiative.

Introduction of a new programming language will inevitably involve some blood-letting. But the transition might perhaps be accomplished by a short and sharp clash with the palace guard (old guard) rather than by a civil war involving the rank and file. The most conservative elements in an entrenched language culture are probably the middle management. In order to facilitate change to a new language and methodology it will be necessary to retrain or replace the middle management and to

allow younger people within the culture who recognize the need for change to take a leadership role. A culture must not be viewed as a monolithic black box which can be expected to respond to high-level directives. We must understand the internal structure and social dynamics of the culture we are trying to displace. High leverage points within the structure must be identified and pressure should be applied so that a minimal amount of radical surgery is necessary.

Three important factors in introducing a new language are documentation, training and economic incentives. A good programming language manual is necessary but not sufficient documentation. In order to convince programmers that the language is appropriate to their needs and show them how to use the new language, there should be well-documented, well-written sample programs (benchmark programs) in each of a number of application areas. Such sample programs would allow instructors to introduce the language in a meaningful way by example programs rather than by teaching language rules. Presentation of the programs would automatically introduce trainees to good methodology and documentation standards in the new language and prevent the decline in productivity due to imperfect understanding or resentment that often accompanies the introduction of ambitious but complex new products designed to increase productivity in other areas of human endeavor.

Learning by reading well-written, well-documented application programs is an effective mechanism for technology transfer. It has not been possible to use this technique in the past because of the difficulty of producing well-written sample programs. However, such programs could produce considerable dividends in catalyzing a flourishing programming language culture. Documentation both at the language manual level and at the application program level should receive high priority in the common language effort.

Effective reeducation is crucial to the success of the common language project. Changes as fundamental and potentially traumatic as introduction of a new language and methodology cannot be accomplished by a short course given by teachers outside the organization. Leaders within the organization must be identified and given authority, so that cultural changes can be internalized.

Incentives for transition to the new language should be provided at many intermediate management levels. A deep understanding of the structure of an organization, the behavior of managers and of the way they can be persuaded to do things is essential. High leverage points at which the carrot and the stick may be applied are rarely direct cash payments and can be very

subtle in their operation, involving social approval, better working conditions, considerations of status, and brownie points for promotion. The explicit and implicit system of incentives in software organizations should be carefully analyzed and should be restructured so that innovation, imagination and flexibility are rewarded, although not at the expense of productivity and reliability.

In conclusion, I should like to point out the dangers of developing systematic techniques for replacing an existing language culture by a new, supposedly better culture. In following this path we are led to advocate reeducation techniques not too dissimilar from those advocated in communist China. We are advocating systematic and efficient methods for destroying a culture presumed to be bad and replacing it by one presumed to be good. However, if such techniques are developed they must be used with great caution. There are many examples of techniques for changing flourishing cultures getting into the hands of the wrong people. For example, Hitler's objective was to transform the contemporary German culture, which he presumed to be tainted by Jewish and other impure influences, into a purer, quintessentially Aryan culture. Perhaps there is less controversy about the desirability of replacing existing embedded computer languages by the DOD/I culture than there was about what Hitler tried to do. But DOD/I is as yet incompletely defined, and people like Dijkstra and Hoare have voiced reservations that the end product of DOD/I development might not turn out to be as superior as optimists are predicting. Since the proposed changes will indeed be traumatic, culturally painful and economically hazardous to a good many people, we should not rush headlong into cultural change unless we are very sure it is desirable.

If we are not totally sure of the desirability of introducing DOD/I, we could refrain from coercive manipulation of existing language cultures and allow DOD/I to prove itself on a voluntaristic basis. This has the disadvantage that DOD/I would take far longer to establish itself and that the inefficiencies of the present way of doing things would take much longer to eradicate. There are clearly different degrees of coercion possible in introducing a new language culture. Tradeoffs are desirable between coercion which accelerates adoption, and caution which reduces both human suffering and technologic mistakes of the new culture. If a new language culture is imposed from above the new master should be gentle and humane. The easy way out of sledgehammer edicts from above leading to perplexity and resentment in middle management due to insufficient guidelines and fear of obsolescence should be avoided. Even a humane conquest will not at first be appreciated by the conquered population. However, the degree of dislocation and the speed with which the new

culture takes root and becomes effective depends greatly on the educational and sociological mechanisms used in effecting the transition from one culture to another.

The above remarks are intended to justify the title of this session by illustrating in vivid fashion that programming language cultures are almost as real and relevant as cultures associated with natural languages. I'm going to ask Rob Kling to present a second perspective on programming language cultures and Patricia Santoni to discuss ' practical experience in providing technology transfer and culture transition facilities in the Navy.

R. Kling: I have very some brief remarks about language culture. I should say personally my experience with language cultures is limited largely to the LISP community which has been one of the more active and rich language cultures. It also represents a very voluntaristic culture in that there is no mandate that anybody must use LISP. But for some programs it's a relatively attractive language and many uses have developed somewhat voluntarily.

I'm impressed by Peter Wegner's revolutionary fervor. I would like to draw an analogy between developing language cultures and the characteristics of certain social movements. The DOD1 effort is to be a program of social change almost on the scale of the Equal Rights Amendment, the civil rights movement, or prohibition. It entails non-voluntary compliance, across the board, in hundreds or thousands of programming shops with a particular language and its preferred narrow array of programming conventions. It contrasts with voluntaristic movements such as LISP and APL.

There are other examples of voluntaristic movements which have attracted a fair amount of attention. For example, VW use. Apparently during the 50's VW drivers used to honk and wave at each other. There is a small literature of self repair manuals for VWs. And VW use spread, but there was no federal mandate or any other state mandate that people need drive VWs or not drive VWs.

The easiest analogy to draw is between DOD1 and other languages, such as FORTRAN or APL. I think that that is too narrow a basis for comparison. What seems to be on the mind of DOD1 advocates is actually a much broader scale, non-voluntary form of social change. The appropriate analogies may occur from FRA, civil rights, and prohibition rather than from LISP, APL, or COBOL. From this point of view something more than an ecological niche is really at issue. Those social movements that seem to gain support and attractiveness have visible symbols. There are cheap ideology slogans, the kind of things that you wince when you hear: small is beautiful. And successful social movements often have charismatic leaders, the Ralph Naders of the world, who basically focus attention with some fervor. They embody a point of view which others try out and which then draws the interest of a lot of people. I'm not sure who is going to be the Ralph Nader of DOD1. If it is expressed in Directive 2-42-73A it is likely to draw a lot less personal interest than if there were the Ken Iverson of the language.

And it helps to have some social resources. Why would people care about using DOD1 and promoting it in their home organization? One set of reasons may be that it is simply a technically better product. But if use is voluntary and products are often special purpose, then one would expect that the use of the language would mirror that of APL, LISP, or the VW bug. Some people would be excited by its aesthetics or prefer its style. Others would prefer to use something else, even though that something else may be

clumsy in the eyes of other beholders much in the way that Pontiacs are clumsy in the eyes of VW owners. So other kinds of resources need to be manipulated to encourage universal use. For example, the status of a potential language user could be enhanced by offering certificates or awards for DOD1 proficiency.

If you look at the social movements that are most successful, many of them are voluntaristic movements. They draw people who are particularly interested. There are very few who have an incentive to join because they gain access to new resources; they enjoy the associability, new status, or new contacts, etc. But it is not just on the technical merits or social merits alone that people become attracted to large scale social movements. Those that are mandated, and I use prohibition as an example, may carry the day for a short period of time. ...

I don't have good ideas about how to build a language culture for DOD1 that would be guaranteed to be massive and be effective on a large scale. I do suggest that those people who are interested in building such cultures really take seriously the analogy between a DOD1 language culture and large scale social movements of the sort that I am using by analogy, rather than simply the voluntaristic language cultures that we associate with different programming languages like APL, LISP, etc., because these latter cultures are much more voluntaristic than the DOD1 enterprise is meant to be.

F. Taft: Would you say that the large scale social migration toward Pascal languages in general is going, in some way, to support the adoption of DOD1?

R. Kling: Probably. The acceptance of VW's also supported the development of Japanese sub-compacts and the movement of Americans to compacts. Some ideas pave the way and other variants of them are more likely to spring up and be accepted. Beyond that I'm not sure what could be said. Pascal use is still voluntaristic.

Col. Whitaker: On the voluntaristic side ... what you may not have factored in is that the prohibition amendment has already been passed. The first step in the exercise was to get DOD Directive 5000.29 which prohibits, if you wish, the use of assembly language, which is the competitor. So we have passed the amendment and now it is just the enabling legislation that we are worried about.

And we hope that it will be a long term movement. On the level of the people that are making the decisions, it certainly will be a long term movement. None of these languages are voluntary on the part of the individual programmer. He obviously has to work with whatever system he is given, whether it is in a local university computer center as the only compiler they have, or whether it is mandated for a DOD system.

[Unknown]: I can think of a couple of incentives. The first one is that if there are superior tools available with DOD1 and the second one is if a wider group of target machines is available.

R. Kling: I have some data which may shed some light here. It's not about programming languages, but about the transfer of software applications. One of the areas that I have been studying the last few years is computing in the local government agencies. Local governments often place a lot of value on making programs portable, for example financial accounting packages or land use models. There are economic arguments about saving the cost of redevelopment that make it seem attractive for agency A to pick up the package developed by agency B. There are also certain technical characteristics which make it easier to move a package. If it is [written in] a somewhat machine independent language, if it has good documentation, all of those things would seem to make it easier. They decrease the burdens of moving packages from agency to agency.

If you actually go out and look at the extent to which local agencies transfer application packages, it is relatively negligible. Most application packages are developed in house. And part of the reason for that is that there are tremendous local incentives for people to develop their own thing in house. For example, it is more fun for a lead analyst to be the head of his own new development project with its own new acronym and its own new fleet of programmers than it is to be the head of a quick transfer job. It's easier to be the expert, and it's more fun to be the expert on one's own system than it is to learn about somebody else's system and try to adapt it.

It's easy to say if we develop systems so they are technically easy to transfer with machine independent code, with good documentation, that that would provide an incentive. These are not "incentives." Similarly, those people who suggest that a rich set of software tools or many host machines for DOD1 provide an incentive to use DOD1 have developed a faulty analysis. Those are not incentives; they simply lower the cost of using DOD1. Incentives have to do with kinds of things that people get their kicks out of -- career mobility, additional perquisites, etc. If there aren't changes in the well being felt by the managers or programmers who use the tools on a day-to-day basis, all you have done is lower the cost. You haven't increased the incentives for DOD1 use, and DOD1 use may well not increase very much.

[Unknown]: And so you think what I said is invalid?

R. Kling: It's not invalid, you've suggested a helpful but not sufficient condition. Increased richness of tools and target machines help, but you can't stop there. I think that if you look at software application transfer you will find that it is an interesting analogy.

N. Finn: What you just said is true now, but there also seems to be a lot of pressure on local government to transfer those programs. I think that as people start being able to transfer those programs, your head analyst is going to begin to see that when 10,000 have written an inventory management program it is no longer so glorious to write another one. If he can transfer this program from somebody else in a lot less time, he'll have time to do something a lot more interesting and at a lot higher level. People are beginning to realize that, too.

R. Kling: We don't see that reflected in our data. And we've got good data on transfer in and transfer out from local government settings. I can really speak in this case from solid data from 500 different American cities and 500 countys.

N. Finn: Do you get the impression, though, that it is changing in favor of transferring software?

R. Kling: There is a lot. Local government staff visit other sites to see what they have done. They then go back and "improve" other applications in their own district based on what they have seen.

N. Finn: When you say "improve" I assume you mean really discovering the same mistakes.

R. Kling: Sometimes people learn, but what I am saying is slightly different. The incentives in one's own organization are to create new systems locally. Local government staff like the arguments about "marginal differences." For example, "the city of Irvine just isn't the same as the city of Costa Mesa, our purchase orders are different." I hear those same arguments about embedded systems programming. "We really need access to peculiar feature Y which isn't available in language X. And by gosh, if we don't have feature Y then we can't use language X, and it is really clumsy to use language X in the absence of this peculiar feature." People are very adept at making up such arguments when it's in their interest to do so. There is very little I hear in the discussion of NONI development that makes it in the day-to-day interest of many managers or programmers to really dive into the language in a major way.

Col. Whitaker: What you've got to watch out for there is that the services themselves are very strong. So in that sense DCD1 use is voluntaristic.

I might point out that there is another thing that may offset the analogy that you have there. The person [R. Kling describes] who makes the decision about the applications package to transport or write is the head of dataprocessing or whatever in a company or local agency. It is therefore to his personal benefit that he gets to do this wonderful thing that he wants to do. The person that makes the decision of a similar nature within the SPO doesn't have the same rewards. Software is seven levels down. He is

lucky if he can even find it in the accounting system. So he makes decisions on the basis of his total cost picture, and not on personal ramifications.

R. Kling: You make it sound very rational.

Col. Whitaker: That's the way the system is organized for sound reason.

P. Wegner: Concerning the seven levels between [SPO management] and the programmer, is there any reason for tinkering with any of these levels or providing incentives that are special to a specific level?

[Scattered remarks continue. P. Wegner then introduces P. Santoni for her presentation on technology transfer.]

P. Santoni: Let me start by saying a little bit of what the basis of my experience is. About two and a half years ago, in a joint Navy and ARPA project, we decided that we wanted to create a thing called an SDL, which is a Systems Design Laboratory, the idea being twofold.

One is to make available existing tools. Consider the situation that's in the Navy for programmers who are really working in a very primitive environment. The system designers and developers are working hands-on doing self hosted compiling, etc. -- all the big bad things we have been talking about. There is very little knowledge of any other tools around. Perhaps some of them have heard of HIPPO. For the most part, if they have tried to pick it up, they decided it was no good for one reason or another. So you've got a very primitive environment.

There are tools that exist in the world. The number one problem is, of course, that most of these tools are not geared to the military programming languages. Someone has to be willing to go out and procure tools that are geared to CMS2. So that's one problem. But there are a lot of tools in the world and the technology is there to provide them to people, if you would just put them into an environment where the people could access them. Clearly you can't put this whole host of tools on the UYK-20, so we started out in a widely accessible environment. We choose the ARPANET, putting up all the tools we could find that were available or that we could promote for the military programming environment.

The other part of our policy addresses the issue that there are a lot of tools that do not exist today, or there are areas in which the research is still at a very rudimentary level and there aren't widely available tools of any kind. In the area of test and V and V, this is somewhat better, but especially in the design and requirements areas, there aren't an awful lot of real computer-based tools you can put your hands on and offer to people. So the other thrust of our work is to promote research in those areas. I consider my function very much one of technology transfer.

There are a lot of people out there in the trenches, so to speak, and they come in different personality types. Some of them are aware that there has to be a better way to do things. They have hit their heads against the same problem too many times, and they say, "if only somebody would give me something" -- or maybe they don't even know what they need -- but they just wish somebody cared. They don't know what door to knock on. Those kind of people, if you can get the word to them, will come to you. Then there is the opposite end of the spectrum, the people, who even if you hand them an assembler, wonder why. Believe me, I have run into this type. So you've got a whole range of people.

A couple of other background things before I go into more specifics. You have to realize that basically the military and civil service is a conservative organization -- middle management especially. Risk is not something that is attractive to them. They would rather pay more than take a risk. And a risk is not necessarily a chance of failure; a risk to some of these people is a chance of overwhelming success. They are just not willing to disturb the status quo. Some of the people who get promoted are those who rock the boat the least. You can point to them and say, "but he's never done anything." Right, but he's never disturbed anything either. Especially with the middle management kind of people, you have to understand that they don't want to disturb the way things are done now.

- P. Wegner: How about fear of obsolescence? Does that come into it?
- P. Santoni: I've never seen much fear of it. The predominant computer in the fleet right now is 20 years old. It's not one of those things people worry about.
- P. Wegner: If you go to DOD1, if it's going to make a big change, might not resistance come in part because of fear of obsolescence?
- P. Santoni: Could be. There are people whose empires depend on it. You were talking about incentives. Lack of risk is an incentive. Low cost may be an incentive, but you are all aware of all the vast cost overruns that have happened without destroying significant careers. Another thing is the idea of an empire. If you are talking about a middle management incentive, a lot of these people are oriented toward the empires they can build. And an empire may be that he has all the people that know anything about NTDS, or he has all the people who do all the radar work in this corner of the world. Empires are based on all kinds of really strange things, but I can imagine a DOD1 empire at each of the Navy laboratories for instance. Who knows, that might be the kind of incentive you can give people, if you want to.

Another thing to keep in mind is that what we are trying to do is create an environment. The idea with DOD1 in particular, and with the whole thing, is to improve the

production of software in the military. And, I think we all accept that you can't do that with just a language. When you are introducing the idea of "life-cycle cost", you are trying to change a situation of small development costs and huge maintenance costs to one in which more money is spent on development. Well, when you are talking to the guy who is funding the development phase, you are trying to tell him that he has to spend twice as much money so that the guy next down the line can do a cheaper, better job. He is still going to come in with the lowest bid, and the lowest bid perhaps is going to be the one, on paper anyway, that just uses the minimum set of tools and doesn't do all this other fancy stuff. So you have to look at how you influence the people that are above that developer/maintenance level. You have to influence the guy who is funding the whole mess. I frankly don't know where you find him in the Navy, but he must be somewhere. He has to locate those project people and convince them that life-cycle is their concern and they in turn should issue policies that say "developer, realize that you do have to put in larger costs so that we can lower the maintenance costs."

[Some further discussion of incentives ensues after which P. Santoni continues her remarks.]

... One thing I would emphasize is that you have to build up user confidence. In particular you have to have tools that work. When you deliver something, you have to be sure that it's going to be ready and workable.

The next thing is the tools must be usable. In my environment I'm talking to projects who are used to doing hands-on work on a UYK-20. They write their code on a coding sheet; they hand them in to keypunch operators; they take those decks of cards and put them into their UYK-20. They work the maintenance switches and control lights, and they debug that way. This is their whole cycle. Self-hosted compile, self-hosted load, everything is in that environment. I'm trying to take this poor guy who knows hopefully everything there is about the UYK-20 and transport him out of that environment and set him in front of the terminal and say, "now do all your things." First of all, even working a terminal is probably strange to him. He knows how to do business now, today. You have to make it attractive to him to do business in a more modern way.

Great documents are not enough. I have handed users stacks of documents and they come back to me the next day and say, "I tried to use the system yesterday and thus and so doesn't work." "Oh, it doesn't? I was just using it this morning. Let me see what you were doing. Oh, here it is, this page in the manual." "Oh, I see, it's written up in the manual, is it?" They don't read them. Typical attitude. So, your key is people then. I spend three-fourths of my time being on-call. You must have people with thorough knowledge of all your tools. Those people have to be readily available; they can't afford to be snippish kinds of consultants. But you must have that people element in there because

otherwise, what you're doing is taking the user out of a familiar environment, throwing him into a strange pond, and saying "now, sink or swim." You can't afford to do that in the DOD or at the worst no culture will grow up at all. At best perhaps several individual cultures will grow up and the individual set of tools and everything. There's got to be a very strong force in the DOD that says "we're here to help".

Now that you have your tools absolutely ready and you have your perfect consultants and everything, how do you get people to come to you in the first place? There are lots of different incentives. In one case, a group came to me mainly because they had been mandated to use CMS-2 and could not afford as an individual project to purchase their own compiler. I had a compiler. I've got something they don't have that they either have to use or want to use. So that's one way to get people to you. Another thing is to have a super-duper tool that is superior to anything they've got.

[P. Santoni describes an example of a UYK-20 emulator that she offers to users as an incentive to use the SDL system.]

... There are also obviously other ways to do it. There are management deals. Washington-level, sponsor-to-sponsor-level armtwisting is possible.

[P. Santoni also describes certain contract incentives that can be used to induce people to use a particular language or system.]

... The next thing I'd like to talk about is the personalities involved, which I hit a minute ago. I believe it was you, Peter, who mentioned trying to open things up so that the younger people can get in. Very few of the programmers whom I have run into on the projects that I've been associated with are people out of universities -- very, very small percentage. Most of them have been around for several years. However, that does not mean that I have not seen the bright, new personality type who wants to use the tool. So I would take issue with whether it's a matter of age. It's more a matter of viewpoint. It's more a matter of finding those individuals, regardless of their background or age, who know they need something and as soon as you offer it to them they'll try it.

[Some further discussion continues in which P. Santoni describes her personal experience in helping people to use the SDL system. She reiterates the importance of providing incentives for all levels, from management to programmers.]

H. Stuebing: I can tell you that it's very important to have your management's endorsement, but unless industry accepts it as well, even that management's endorsement won't be enough. Ultimately what it comes down to, in a schedule driven deadline situation, is the acceptance by industry.

If a company stands up and says, I can't get that job done that way, I'll tell you they won't do it. No system project office or project manager will take that risk.

- P. Wegner: Suppose DOD1 becomes accepted and you want to accommodate us in your shop, what would be involved? How would you go about it?
- P. Santoni: Right now I have the environment where I'd be more than happy to be the original site of any tool anybody wants to bring up involving DOD1. I would except that when DOD1 compilers come around I would get one of them first.
- P. Wegner: Would it be appropriate to have several or many efforts like your own to do this?
- P. Santoni: Right now the SDL project is still a research project. It's the idea of exactly what we've been talking about here. How do you integrate all these tools? How do you make a really usable user facility that has this huge range of tools? There is a proposal which, as far as I know, has not yet been acted on, that says there should be at least a half dozen sites equivalent to what I've got right now. In our case again around the ARPANET, but that isn't even a necessity. So there should be several geographically expanded or distributed sites for something like this. ... It's very important that the DOD1 be willing to set up these kinds of shops.
- K. Bowles: I'd like to make a comment. Some of you know that we have a stand-alone software system based on Pascal that runs on small microcomputers. We've been observing a phenomenon that I think probably could be brought to bear on this spread of DOD1 in trying to spread our own system and spread the use of Pascal. And I think it would at the very least relate to those programmers who work with contractors. I can't really predict how it would impinge on programmers who work for the military agencies because of the slightly different groups of people. But in an increasing number of cases we're noting that where commercial firms have decided to commit to the use of our system that it's often happening either coincident with or possibly slightly after the individuals on the programming teams have decided that they can afford to go out to a computer store or an equivalent place and buy a machine which is able to run our system. So these professionals are taking home the Pascal-based system, learning to use it and love it, and then have enough information about it to bring it to their work environment.

Within a couple of years this system of ours is going to be available widely on machines costing in the \$1,000 to \$1,500 range, if not less. There's no reason from an engineering point of view from all we know or are able to predict at this point about the DOD language, that the same language couldn't be supported on only slightly larger machines. So if it's a better language you ought to be able to build upon the class of people that we're already reaching or will have reached by that stage and quite possibly take advantage of

that phenomenon. I think it's non-trivial phenomenon from what we've seen.

[Some discussion ensues regarding the technical feasibility of hosting all of DOD1 on a microcomputer. No firm conclusions are reached.]

W. Loper: One of the things that we had in mind when we were thinking about this topic was some form of informal users' publication and program exchange.

P. Wegner: Like the Pascal newsletter?

[Unknown]: Or Interface Age.

P. Wegner: As a matter of fact, I think you should consider having a newsletter that initially has some funding support in connection with DOD1 activities. That could be done from the top very easily. The DOD1 News -- it's an activity that needs to be arranged. It would be very worthwhile and you should perhaps consider starting it within the next six months.

N. Finn: Between the newsletter and the P.R. campaign and the support teams and the generation of all the new tools and everything, I think Col. Whitaker could build himself quite a nice empire!

P. Santoni: Given that list that he just enumerated, I am not sure that DOD is really cognizant of the large dollar investment that's going to be necessary to make this thing transfer. I have no idea what the DOD1 budget looks like. It's one thing to talk about what it's going to cost to build compilers and to copy those things and send them out to various hosts and how you persuade contractors, for instance, to write tools for the language. It's a whole other thing when you're talking about the technology transfer and you get the whole people issue -- that's dollars.

P. Wegner: Col. Whitaker, has there been any projection of the cost of the technology transfer? Will you tell us a little about that?

Col. Whitaker: It's very carefully buried in a 1.2 million dollar per year item, actually.

P. Wegner: You say it's buried within 1.2 million dollars. We're talking about ten or twenty million dollars at least, aren't we? Could you just enumerate those activities again?

N. Finn: We had ads, demonstrations, support teams, tools that work.

P. Santoni: You're talking about equipment too. [Also the ARPANET.]

P. Wegner: I think it would be appropriate to identify this

technology transfer activity with a statement of what is to be accomplished.

[P. Wegner solicits volunteers to draft such a statement and the session closes.]

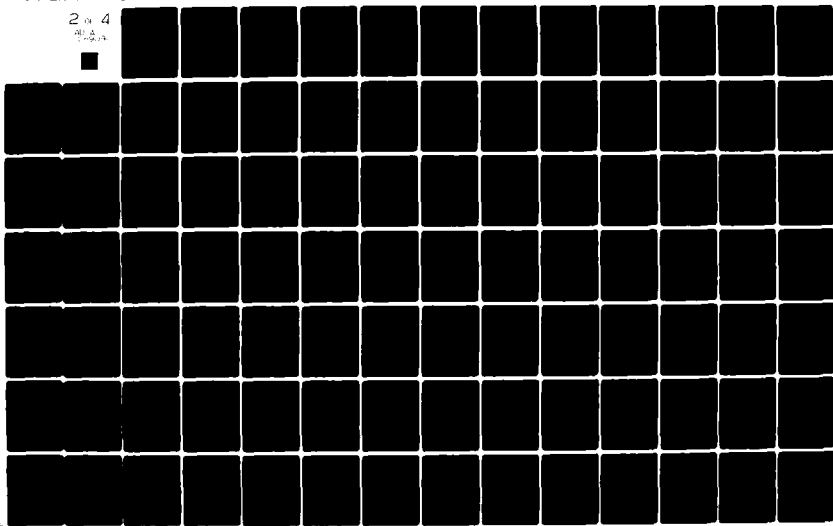
AD-A089 090

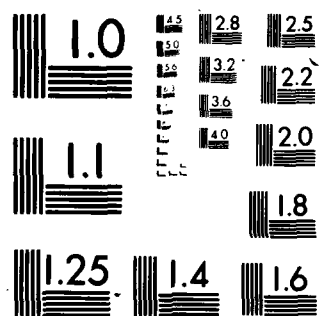
CALIFORNIA UNIV IRVINE DEPT OF INFORMATION AND COMP--ETC F/G 9/2
PROCEEDINGS OF THE IRVINE WORKSHOP ON ALTERNATIVES FOR THE ENVI--ETC(U)
1978 T A STANDISH DAAG29-78-N-0219
UCI-ICS-78-83 NL

UNCLASSIFIED

2 0 4

20 4





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

Session 1B: Requirements Analysis
Henry Stuebing, Chair

- H. Stuebing: Let me begin by reading paragraph 7.4 on Requirements Generation Tools from the preliminary DoD Common Language Environment Requirements of 15 May 78. "A method of cataloging system requirements in order to test throughout development for requirement compliance will be provided. This capability will be similar to that of the Problem Statement Language/Problem Statement Analysis programs (PSL/PSA)." The only other system that I am aware of is the SREM system that TRW has developed. That is just one computer system in an Army application. Keeping in mind that as a Workshop what we would like to do is to stimulate a lot of discussion and get any kind of thread of consistency amongst opinions. Some of the things we might talk about are: What is a Requirement? What does it mean to do an analysis of a requirement? What are people's thoughts about Traceability of Requirements? How should requirements be relatable to different statements in the programming language? If you are going to trace a requirement all the way down to some piece of code, how should one do that? What are some of the mechanisms for tracing? Our goal is to stimulate your thoughts and get ideas flowing so we can take these two original sentences and give them more meaning in the final document.
- J. Machado: Can you define requirements analysis?
- H. Stuebing: I don't know if I can define it. What I have seen is people coming up with different approaches to state their requirements with some degree of formality such that requirements are storable in databases, and things like that. Then using various techniques, or using the machine itself to do an analysis, or testing for completeness.
- J. Machado: Today's technology does not allow us to test for completeness of a requirement.
- H. Stuebing: That I think depends on the assumptions that one makes. Why don't you tell us?
- J. Machado: I am trying to stimulate some discussion on the state of the art in this area. I think the suggestion that PSL/PSA should be required is not the correct solution at this time.
- : The requirement on the system is the relation on the system element that is required in order for it to be consistent. Relations are mathematical things defined on sets.
- J. Machado: Here, we are concerned with requirements analysis in terms of DoD, (if we talk in terms of the requirements of the tactical systems). Some questions to ask are: how do you specify the requirements of the tactical system, and how can you trace them throughout the program to insure that your final product meets the requirements that you've

listed? In the first place, how do you get your requirements accurately specified so that you're getting what you want? You want to make sure that you have in fact a complete set of requirements. How do you find that out?

-----: It depends on your [high level] requirements and your requirements on a program.

- A. Irvine: I have a problem with this definition. Our terminology says you do some analysis or do some requirements definition to define the requirements. It is much more concerned with the relationships between the system and its environment than among system elements. Design begins to consider the relationships among the elements of the system.
- H. Stuebing: Let me propose a problem that we may run into. It is a fact for any level in the design process, the level above it is the requirement in terms of the level that you're working on. This is just inescapable all the way back through the system process. This is why I brought up the question of what are we talking about when we talk about requirements analysis. I think we have to define what level of the systems design process we're going to hold our conversations to and then talk about what applies in the way of supporting environments. I don't think the Workshop can proceed any further until we have common understanding as to what we mean when we say requirements analysis. What level of this system design process are we talking about? Are we in front of the actual design. Are we right at the beginning stages?
- R. Balzer: But Al's definition still holds if you take the right interpretation of the environment. That is, as you get further into the construction process, your environment shrinks. More things are outside of where your concern is and therefore are fixed and are no longer under your control. The requirements have to do with the unspecified part of the system. You'll be replacing those things with a method of attack. Once you do that, your concern goes into a smaller shell where more things are fixed and you have a smaller set of things yet to define. I think the notion that the environment is the place where the requirements deal with the environment is quite appropriate. You are taking things out of that environment. You are defining things and placing them outside your scope as you go down into successively more detailed levels.
- H. Stuebing: We have frequently done that and it is simply in terms of that context. The requirement for the next operation is a carefully bounded context.
- R. Balzer: To put it in terms of DoD technical development requirements for the highest level document: "How do you analyze requirements to make sure that you are doing what you want, and how do you trace them through the system?"
- A. Irvine: I suspect a more common meaning is that of "initial requirements," which defines the requirements of the whole

system.

H. Stuebing: I don't think that's the level here. One can get into the operations research end of the game where people are challenging the need for the system. Rather, it says, "systems requirements" in the sense that it is given that this system is needed. There is some level of statement from an overall fleet or service standpoint that says it is needed. Those documents are very high level. Now, we are going to build this system. This specifically says system requirements. As soon as I see the word "system", it means to me hardware, software, man-machine interfaces --- all those things.

J. Machado: I would argue that you don't get into hardware/software tradeoffs at the requirements level, those are implementation issues.

-----: You want to make sure that the set of requirement processes that you go through say that this is the functional part of the system, that it is consistent, and that it is complete, i.e., that you haven't left something out of it and further that you haven't been redundant in the specification of the requirements.

-----: It also implies tracing to make sure throughout the system that the requirements are met at each level. There is something missing from this discussion. Section 7.4 says "test throughout development for requirements compliance". That suggests to me, as someone mentioned, that each person's requirements are imposed by the person above him. We have got to keep in mind that we're talking about requirements all the way down to the lowest level which is a nested set of requirements.

R. Balzer: No, I disagree. What you are saying is you trace requirements. The traceability is that which you test at every level to make sure the requirements of the higher level are embodied in that level of design.

A. Irvine: The only way you can do that in practice is to treat each level as the requirement for its successor. You test for consistency with its predecessor.

-----: I don't disagree. I think that, as stated, there is a hierarchy involved with the requirements. I think you have to start off at one point and define what your problem is, and then you can iterate that particular process down through the design. But if you start to try to attack from the iteration, to start off with, you can get lost in a maze.

H. Stuebing: Are there experts in the room on PSL/PSA? Perhaps we can learn what is the state of the art in formal language specification requirements.

A. Irvine: I know something about it. The most significant characteristic of it, as far as I'm concerned, is that it is

a narrative language. To me that is important. It is essentially a language which has assigned a set of interesting verbs and nouns. You can observe interesting characteristics about those key words. For example, it assumes that if you tell it A uses B, B will be used by A. That's the basic nature of the way the analysis is done. You can write hierarchical definitions, very much like you can in a nested programming language. It produces formatted reports.

-----: Why is it not sufficient at that upper level?

A. Irvine: I have used it a little bit. I have seen it used in system documentation. I cannot visualize what is going on in the design from that kind of documentation.

R. Balzer: Basically PSL is a dataflow model of a system. It tells you how data originates in some models and flows to other models and is used as inputs and outputs. It worries about the interconnectedness. It really is a flow chart of data. The thing it does not address is what the interpretation of any of that data is. It doesn't tell you how that data was created. Functionally, it tells you where it was created, i.e., in what module, but it doesn't tell you how, and it doesn't tell you what the interpretation of that data is. That is, what its semantic content is in any form. That's why it doesn't address a lot of the issues that are very important in requirements analysis where you are wondering whether or not this fulfills some requirements because you don't know what it is. You just know that there's something there, that there is water flowing through this pipe. You don't know anything in detail. That's the reason for Al's comment on its being very useful in maintenance, where it's very important to know where data originates. PSL/PSA tells you all that but it doesn't tell you how it is achieved. It's one cut at the issue of defining what goes on in some complex system. Lots of people have recognized that we need something more sophisticated but there's something that exists, it's proved useful, and we have to sort of go beyond that now.

P. Freeman: Let me add one last thing to that. It is a documentation tool. It is a language and an analyzer that helps you express certain things used for requirements. It does not tell you much about how to come up with those requirements, or to check semantically whether that is the right set of requirements with respect to the outside world. It is not a methodology for developing requirements.

R. Balzer: It does catch certain kinds of inconsistencies in the specification. For instance, it does do unit checking on types. If you have an output in feet per second and you need it in miles per hour, it recognizes that there is an inconsistency and tells you where the sources are.

-----: I would like to give another view of it --- more of a generic view. It is a database oriented system where there is a mechanism for recording the information in a database and taking information out of the database in forms

that are useful to people. The basic concept of the database is identifying objects that the user is concerned with in his application, attributes of those objects, and relationships between one object and other objects. That is the basic underlying structure. If you are not happy with your narrative language in PSL/PSA, you can construct your own narrative language, particularly suited for a particular application. So to follow up on your your comment about its not having the semantic information, it would seem to me possible to add descriptive attributes or even paragraphs that you can put in during the machine process. I think the significant thing is that it is a way of recording information that is easy for people to interact with and it describes the system from the point of view of the user.

- J. Prescott: The intent with the environment document is to define the environment for DoDI. We are trying to make a successful project implementation of some set of requirements. Perhaps we should have stated in there "software system" as Hank said earlier. You are given some statement of work, and you want to translate that statement of work, which has your requirements in it, into some functions called programs. At the end of that whole process you want to be sure that you have built what you have contracted for. You build back to those requirements. You would like to have, as you are going through the development process, some way of knowing why you are building a particular piece of software. The reason you are building this piece of software relates back to requirement 3.2. Now recognizing there is no one tool that does all jobs in development, you still have to pick some. So you pick something like this Problem Statement Language which helps you define what the requirements are, and the Problem Statement Analyzer, which takes that language requirements definition down to another massive set of programs, and gives you some help in finding errors. Fifty percent of our problems today in building software are design type problems. You want to find them early. This type of tool helps out. Picture the environment for construction of programs. We have many tools, one of which is some technique which starts from the statement of work and traces all the way down like through a big matrix. Like here are the things that you must do with this software system, here are the programs that you have built. Now picture some checkpoints in here. This program fulfills requirements here, and this program fulfills these other requirements. Think of the beauty of this as you go along because requirements have been known to change half way through the project. What you would like to have is a way of knowing which programs are affected by these changes. This particular set of programs that I mentioned in the document was developed at the U. of Michigan. There are a number of other types of programs that do very similar kinds of things. I put a very specific case in the preliminary PEBBLEMAN document because that was the one I was most familiar with. Understand what we are trying to accomplish. We are trying to find not one big tool, because there is not one, we are trying to find all those tools that we need to help us in every place along the development cycle.

- H. Stuebing: In one of the related papers by Dick Taylor, from Boeing Computer Services, Dick has a couple of diagrams that help summarize these points.
- R. Taylor: There are two of them attached on the back of the position paper. These capture some of the things that have been mentioned. You have all seen the classic waterfall chart, and you probably realized that we rarely have a "true" waterfall. As requirements change or design problems are discovered, you find the need to go back to a previous phase in the program development. The thing that I find interesting is this picture (Figure 1). The thing that we are doing in this box is making some formal statement of what the system should do (Requirement Analysis). The things we need to do at that point are: One, check that the statement we have made of the system is internally consistent. Two, we need to check that we do capture the user's need. Then as we go on to refine the problem solution in preliminary design, the same situation happens. We probably have a different external representation of our solution, but we need to check to make sure it is internally consistent, and check that it goes back and really does satisfy the requirements statement. The same thing happens in detailed design. I am most familiar with that kind of checking being done in coding. We have static analyzers and symbolic executors and so forth. Most automated tools have been directed at code. It's Boeing's view that these kinds of analysis should be done in all stages. The interesting thing is that the nature of the analysis is the same in each of those phases. Only the external representation of the solution we have chosen has altered. So, we can conceptually think of having a requirements language front end, a detailed design language front end, and so forth, all forming some sort of common representation in the program (Figure 2). On that we can do the verification we need using whatever techniques are appropriate. For example, it might be really appropriate to use symbolic analysis and formal verification at higher levels, like requirements, since we don't have the level of detail there that we do when we get down to code. By the same token, it's hard to do any kind of dynamic testing on requirements since you don't have an algorithmic specification. There is another item I want to mention. Something that was referred to earlier, is the program database. I think we ought to keep in mind that one of the big problems in creating software is providing appropriate management. That is the whole motivation in doing a lot of work in requirements in the first place. If you get a really rigorous specification of what you want to do, then you have some ability to see whether you are solving the right problem or not. The database is a common repository for everything known about a developing system. Further, we haven't used PSL/PSA to do this. Boeing has its own tool called SAMM (Systematic Activity Modeling Methodology). There is justification for that. The statement I have heard about PSL/PSA is that, as a requirement specification language, it makes a good detailed design language. But SAMM attempts to really do requirements analysis.

- J. Prescott: The word verification is a little bit tricky and I have the feeling that it means something different when it sits under the word code than when it sits under the word requirements. Would you say a little bit more about precisely what you mean by verification?
- R. Taylor: When I use the word verification I don't usually mean it in terms of formal proof of correctness. I would say verification is showing congruence of one step to a previous step.
- S. Crocker: You made some reference kind of quickly at the beginning where it seemed to me you said that these formal techniques, symbolic execution and formal verification, might be more applicable at the requirements end than at the code end.
- R. Taylor: Right. The problems we are seeing with symbolic execution, (and I can't speak for formal verification) is that when you apply them to a bonafide program, you have such an enormous amount of detail that the processor that is doing the verification just get's bogged down. It's not a very efficient technique. Whereas at higher levels of program representation, say the preliminary design, there is less detail, and at that point you may have more success in the use of symbolic execution.
- J. Machado: I believe we should request that some tools be developed to model the requirements. I think you have to present some type of a picture to the user as to what type of system he is going to get. You can do this with a modeling tool. He can get some kind of an indication as to how this system is going to respond to what he has submitted it to do with his requirements. You will get some validation that way, that the requirements are what you really meant. In most cases, we end up writing a set of requirements which aren't what we meant.
- E. Nelson: I think that is terribly important because while reducing the requirements down to paper makes the user look at and read them, he may not comprehend fully what those statements mean. He can say it represents a complete list when it doesn't. If you produce a simulation in your model and he looks at the results, he can see that that's not right or good or that he's getting the wrong answer for the wrong kinds of things. Then you determine what is wrong with the requirements statement. Modeling and simulation catches that early instead of waiting until the program got written and then seeing the outcome is obviously wrong.
- R. Balzer: There are two things I would like to say. First, we have only talked about one kind of requirement so far. Essentially the logical processing requirement. We have not talked about performance requirements and, in general, they tend to be more difficult despite the difficulty we are having with logical requirements. The other thing is that there's been a thread running through a lot of the comments made that's important to capitalize on. Basically the same process is being repeated over and over during the

development of programs, from the very beginning of the requirements down to the final coding. People are taking what is conceived of as non-procedural at that level and substituting a method for achieving it at the next lower level. If you think of a hierarchy of logical machines, then the objects you use for describing things at one level become non-procedural down at the next level and you have to go through further expansion. If you take that kind of view point, I think that one can make an awful lot of order out of this whole life-cycle development process because you are doing the same thing over and over again with the same kinds of tools becoming useful at each of these stages. Essentially, you can model at any level in this process by having an interpreter of the machine that exists at that level. That interpreter may be extremely inefficient, but you are willing to pay that price to get an idea of whether the system is going to behave as you want. Also you can do your tradeoff analysis.

- S. Crocker: I have a very strong theoretical agreement with you. When I play that back against any sort of real life experience, there are a couple of issues that come up. One is that although the tools are conceptually the same and may be the same on a theoretical basis, the specifics are so wildly different that one winds up building different tools. The semantics are different at each level. The other issue is that the numbers do really kill you. One can't just sort of wave one's hands and say that at any efficiency we are allowed to do that, because what you need for debugging code at a very small low level needs a certain response and what you need for debugging your requirements at a very high level also needs a certain kind of response. The mechanisms, the interpreters and so forth, have not so far been able to come through in all these different areas. Also, these underlying semantic description mechanisms that we have are kind of different in all these different levels. So although there is kind of a theoretical appeal, and I believe that the pattern is the same, I think we are quite some distance away from demonstrating that the same tool will be applicable at all these different levels.
- J. Machado: I agree with Steve that although the process that you go through in requirements analysis design, and specification look alike at each stage of the process, the tools that support them are indeed different. At some stages, you want analytical tools and at some stages you want simulation type tools.
- A. Irvine: I will try to summarize some things about SADT (Structured Analysis and Design Technique) that are important in the context of this workshop. SADT places a great deal of emphasis on how one arrives at the resulting requirements, and on the thought processes that are used and how those thought processes are audited. A lot of emphasis is placed on human factors, e.g., how, through commenting and critiquing, you manage to improve the quality of the requirement specification. The significant characteristics of SADT as a language are that it's a graphic language and that it produces models. I argue that the reason the

graphics are important, is that in the analysis of any complex system, what you really have to focus your attention on are: the parts and pieces of which it's going to be made, how you determine which are the appropriate parts and pieces, and what the relationships, interfaces, and constraints among those parts and pieces are going to be." It's my experience that only if given some well defined pictures, can one review with any kind of understanding what the statement of requirements is. So what we do with SADT is construct a few rules which carefully constrain the kinds of pictures we can draw so that the pictures are readable and understandable. These pictures are distinctly constrained diagrams. They show constraining relationships among the parts and pieces. It's only by seeing those constraints in at least two dimensions that anyone arrives at a real understanding of what the statement of requirements is all about. I should also point out that we do represent in these analysis all kinds of constraining requirements, the ones I mentioned earlier: performance constraints, manpower constraints, use of equipment, use of programming languages, etc. There is no detail level that can't be expressed as a requirement. The other thing is that SADT places a lot of emphasis on the thought process. It tries through the methodology to make sure that people who are doing the work are constantly in the process of moving back and forth between analysis and synthesis. That's the essence of getting the job done. You take the things apart and then you see how they go back together again. You look at their differences and then the similarities

- J. Machado: In other words, you are going through a lot of thought process. What about the customer out here, the buyer.
- A. Irvine: I have personally worked on projects where the same document was reviewed by the people who were going to fund the project, the managers in the user organization who didn't know anything about computers, the manager's in the implementation group, the implementation people, both hardware and software, who are going to build it, and the people from the field service. Every one of those people read, reviewed, and approved the same document.
- : Something Dave Fisher mentioned that I hadn't heard before is that if the tools are written in DOD1, they will be more transferrable. How much is that a real constraint on of what we are talking about? Are we talking about tools that eventually would really have to be written in DoD1 before they got into general use in connection with DOD1 programming effort, or is it a possibility that one might pick up some system that's already written in some other language and try to make it widely available still in that language.
- H. Stuebing: I'd like to tell you what my impression is, if I've understood you. The implementation of the tools in DoD1 is not our primary concern.

R. Balzer: Let me argue that case. If you do not require that the tools be programmed in DOD1, then you will have no way of insuring compatibility of environments that exist on different machines. For the same reason that you would like to have compatibility of the language on different machines the compatibility of the tools used for program development is just as important. One of the largest reasons that programs are not portable is because of the environments they exist in, not the languages per se. I think it would be a crime to allow the development of different environments just because you happen to be in a different machine. There's going to be enough diversity because the tools themselves are not going to mesh very well with one another, and new tools are going to be constantly brought into this environment. We should at least maintain some order by having them developed in DoD1.

Session 2B: System Design
Clem McGowan, Chair

- C. McGowan: In this particular area the Pebbleman is completely lacking. It has just a few sentences. ... It says "program verifiers tested in a mathematical fashion used to see that the program corresponds to a given specification in a design language will be required." One of the few things that is mentioned about design is that they're calling out for a design language. Something that may be a corollary to the common language is that there will be a common design language ... It says "design tools required will be those which support a methodical design process". I have a question as to whether they're talking about a specific process or not and feel that the words "which support systematic design" might be better substituted. ... (It says) "Programs will be available to support a design language by testing syntax correctness, formatting, and proof of correctness." Here we have a second mention of verifiers or proofs of correctness and there is an issue as to whether that is to be developed or is it suitable to require it now in the late 1970's. "The design capabilities will be able to compare it to the requirements." Here in very general terms they seem to be talking about something like requirements traceability. ... 7.6.1 says "Methods will be required to translate programs written in a design language to the common language." So we're not only supposed to have verifiers but methods to make this transition from design to code. The final section ... (says) "Integration tools which will assist in fitting the individual programs into a system will be required." ... Something that's not mentioned here under design is specifying a database interface along the lines of the CODASYL group. Should that be part of the language? ... I'd like to now ask for reactions to these specifics and to the desirability of a program design language and an automatic tool to support it.
- G. Fisher: Is this program design language also to be known as a specification language?
- C. McGowan: I would say not. ... (It would be similar to the) Caine, Farber and Gordon tool that supports writing down flow of control with some data declaration capability (which yields) a representation of the design prior to getting to code. You have free formatting and you get the cross reference listings for declared data items. ... It's a design language where you can introduce English language imperative verbs as operators rather than use the restricted operator set of the DoDI language. Pebbleman seems to be asking for automatic support for the use of such a program design language. ... I think a natural consequence of using a common design language like this will be to change some of the standards regarding appropriate deliverable documentation. For example, flow charts may go out the window and PDL may replace it.
- R. Balzer: One of the things that bothers me here is that we're asked to talk about tools for a part of a process that we see as ill defined. We don't know what the boundaries are

that separate program design from specification languages and requirements languages and so forth. I think that there's a fundamental reason why this confusion exists. It exists largely because of manual, historical reasons. Somebody in the past has made a division and we have different languages for describing these various aspects of the construction of programs. There comes a point which you reach at this development where you say "that language is no good anymore" and some other language is much better, so I'll make a block transfer from one language into another. For every language that you pick there is some point at which you reach that boundary and go to the next one. So it's very hard in the abstract without knowing the specifics of the language to understand where that boundary is.

Let me propose an alternative model. ... What I see, and I think lots of other people see, is a lot of commonality in the movement from requirements down to actual code. In the whole span of activities, what one is doing constantly is taking things which are nonprocedural at one level, substituting a method for accomplishing them at the next lower level, and if we could build a language which spanned that entire spectrum so that one was making refinements or elaborations within the language of constructs at one level into the next; and then you could go piecewise between the very high level specification and your ultimate code. You may find that you want to carry some of those activities further; one part of the program is developing before you deal with aspects of the other, and we would get away from this arbitrary translation from one level to the next. You put much more in semantic units, deal with things that you felt were important. I think then a lot of this confusion would disappear if we could talk about on a logical level the kinds of tools we support, the elaboration of nonprocedural aspects of a program. That's what I think the whole process is about.

- C. McGowan: ... I have some difficulties with that myself. For this discussion, I think we should use as our grounding the proposed common DoD language so there's an identified point along your spectrum. It may be that the DoD1 is not extendable upward to be that comprehensive language that allows you to go from soup to nuts.
- R. Balzer: What you're saying is that the bottom end of the spectrum is DoD1 by definition. I'm quite willing to live with that. Consider a language which contains DoD1 as a small part of it. One starts by describing a program using other parts of the language so that you gradually make the translation into things which are DoD1. When you get all done, though you're still in this same language, everything is now in the part of the language that is common with DoD1. There's certainly no reason why we couldn't define things at a higher level which look very much different. For instance, one of the constructs which is extremely useful at the specifications or requirements level is a thing like all X such that some predicate is true.

C. McGowan: I haven't seen many requirements other than in

University settings that have been written that way.

- R. Balzer: All planes which are identified as foes will be tracked ... OK? That's certainly of that form.
- C. McGowan: Fine, we'll accept that at the present. There's considerable elaboration needed for the identification process, including time limits.
- R. Balzer: Absolutely, that's exactly the point. At that level there are lots of things that are nonprocedural without that specification. One gradually has to define all of those things and turn it into an algorithm. In that process of elaboration, you can work your way down to a DoD1 program.
- : I agree with what he said, but a couple of things bother me. I just want to make sure that the process does not become too detailed in one area before other areas. There are problems where you have to combine five or ten requirements statements.
- N. Finn: I don't think that at this stage of the art we can sit here and specify in the Pebbleman as to whether it's going to be one language all the way from the top down to DoD1, whether indeed we can have one language all the way that DoD1 is a subset of, or whether we want a definite hierarchy of languages to keep from going too deep on the top level. I think that we can specify here that we need some sort of problem definition language, and that we need some sort of trail from the top to the bottom. We have to have such a language or hierarchy of languages, and we have to have some sort of automated means of verifying that each level of the design process from top to bottom is indeed an implementation of the design specified by the level above it. If we can establish the correctness traced from top to bottom and from bottom to top that's about all we can do here.

[A short discussion was unable to decide whether a single or multiple level structure of language(s) was appropriate, or whether verification is "a research goal or a reality".]

- T. Standish: I would really like to understand something about the nature of current technology that supports designs in any representation that anyone cares to mention, and what specific things you can do with those design representations, particularly in the nature of tools that help do whatever you can do with them, like check for consistency, or see if they meet the requirements, or map them down into something more concrete, or whatever. What is the nature of the current technology and what does it do for you? Is it any good?

- : I've been involved in one attempt to provide such a technology. [HOS] It was initially based on the experience with the Apollo project. They revisited the Apollo project and looked at what steps they went through in the development of that system. ... They came up with a

requirement to try to reduce the testing at the later stages and bring it up to the front end. In the attempt to do that they formulated certain axioms that one should follow. If you follow them you automatically assure interface correctness. It wasn't a methodology to try to prove this system correct, but it was a methodology of trying to reduce the testing and to show direct interfaces where most of the errors were found in that project. They came up with the six rules, they developed a language which performs to those rules, and graphed a method which also performed to the rules so you could either define your system graphically or use this language called ACCESS, being developed by the Navy. Their methodology has been used on limited projects.

[The speaker goes on to describe uses of the system by machine and in paper exercises.]

- C. McGowan: I personally feel that we want to avoid building a particular design method into Pebbleman. We might extract out whatever we define as useful from HOS.

... I want to return to traceability from the requirements. Should we pin down the context? For DoDI embedded weapons systems being developed, we can think of the starting of design as being the specifications. We'll have something like the program performance specs, and given those in some standard form, let's start the design process from there. A major consideration is the traceability of requirements. Let's get some other case histories.

- P. Santoni: ... There are languages in system design that stand alone as languages. PSL is an example of that. You don't have to have a methodology to use PSL to do anything you want it to. There are methodologies that exist which have no languages. By methodology I mean a set of concepts that says this is how you go about trying to divide up a system. There are projects that combine the language idea and the methodology idea. I think one of the paramount things, especially when you're talking about a DoDI environment for development of programs in this language, is that we're talking about a computer based environment, and I think one of the primary things we might add to this system design tool area is that the tools indeed be computer-based. You take something like AXES; it is capable of being computer processed. ... There's been only the attempt of Hughes to put together structured design concepts with a language. So sometimes you find one or the other. It's a question of whether you have something that you can make a computer-based tool out of. I think that without that computer-based tool, you're never going to be able to do all the traceability things you're talking about. ... That's one of the things I'd like to see emphasized in that section. Computer-based tools.
- R. Balzer: You can talk quite specifically about the kinds of activity that are necessary to translate from one level to the next. You've mentioned some already. There is the traceability issue: that all of the requirements that exist above one level are somehow met in the implementation that

we chose. We need some tool toward analyzing the requirements at the level that we're working at to give us some notion of the tradeoffs between the different ways of meeting those requirements. So we would like some sort of analysis for simulation tools that we could use to do these kind of tradeoff designs. We'd like some tools that suggest different implementations; that is: what are the different known ways of meeting a requirement that exists at this level? That might just be a cook book of useful techniques. Then we would, of course, like a verification that says that this total object that we have created matches the specification at the highest level: that's our feedback rule.

- C. McGowan: ... We must bear in mind that if we have full traceability, full consistency checking, analyzing, etc. , and we have everything in the machine, that we still may end up with a really lousy design.
- A. Irvine: Just as when we deliver a common programming language we are not guaranteeing the quality of the programs. There is still a creative process and just as you have programs of varying quality, you also have designs of varying quality.
- C. McGowan: But the question is to what extent can we provide assistance that will make the design process easier, more manageable, more controllable, that will interface better with this language that will be on the immediate horizon.
- J. Esch: Program design languages depend on what you can express in them to bring out the creative nature of the programming task. What could be agreed on is what kind of information you need to record to specify the design.
... The types of information that need to be recorded are: what the parts are of the program, what the purposes are, how they relate to each other, in both data and code, and then the overall algorithms.
- N. Finn: I think we are going a little too deep there (because) I think there is a great deal of difference between the current methodologies for controlling software design efforts. They have totally different databases. ... I propose that we put in the spec ... a list of items that we all agree on. We have to have traceability, verifiability, consistency at each level, and a computer base. We all agree that we need that.
- J. Machado: I keep hearing software, ... The system design process is by far bigger than the language and the "software". The computer happens to be one component of the system. My first supposition is we are talking about the wrong thing. We shouldn't be talking about software, we should be talking about systems design, literally systems design and not software. The language just happens to be how you implement the software portion of the systems design.

[This starts a discussion of whether or not the software aspects can be factored out of the total systems design

task. C. McGowan points out that some of the methodology developed for monitoring software will be useful in other areas, although there are certain problems that seem unique to software.]

- P. Elzer: ... I really see no difference between software design for better computer systems and the overall systems design. ... It is not wise at the moment to only talk about software; but on the other hand, I think we are giving the language effort itself too little credit or we are too skeptical if we say that the language will not help to improve the design process. What I have seen is that the abstraction technique will have some impact on language design and will be in the language in one way or the other. So it may well turn out that the language itself is kind of a tool which helps to create several abstraction levels above it. At the same time, we should look into hardware and software design aspects.
- C. McGowan: ... My own belief is that there are fundamental differences between software and systems. For example, the fact that systems designs are like wiring diagrams and there's a one to one correspondence between what you write down and what will be built. Whereas in software you can have one instance for example, processes and many recursive replications. So there's not one to one correspondence between the diagram and the realization.

[The participants discuss the need for the ability to handle non-software problems, and to allow for the processing of drawings and other non-software components of the design. T. Standish points out that a text editor and the English language will allow for the capture of a design, but that an important issue is the processing that you can do on the stored representation. C. McGowan attempts to get the participants to make such a catalog.

N. Finn suggests horizontal consistency (i.e. within a design level), and vertical consistency. A. Irvine discusses the need for traceability from the top down (all conditions handled), and the bottom up (everything here is necessary).

Another issue discussed is the desirability of enforcing a top down methodology. It is pointed out that very often the lower level choices are constrained by the desire to use previously developed subsystems.

C. McGowan points out that the ability to reuse subsystems has not been used effectively in previous software efforts.]

- T. Standish: I'd like to return to an earlier question: Is there any current technology that you can identify that has those properties?
- C. McGowan: There are a lot of them around that tell you how to design and in some restrictive cases they work. We can list out things like the Jackson method, the Warnier/Orr method, Constantine structured design, using Bachman's data

structure diagrams, the HOS.

- R. Balzer: These are all guide lines for manual interpretation but there is no tool I know of that given a specification at one level will suggest to you N different designs and say "choose among these."

[The discussants agree that present technology has attempted to process designs via some automated tools, but that there are no tools that assist the design process. Such tools, could, for example, present alternative sets and recommendations.]

- P. Santoni: ... I believe design is a very fluid creative kind of process and it's going to be very difficult in your generic, tactical system for anything, including a human being, to generate all the possibilities. What we do have are concepts. In particular, the methodology that we have been sponsoring, the Hierarchical Development Methodology (HDM), is the closest thing I know of right now that has concepts of design built into a language supported by various consistency checkers, completeness checkers, hierarchy managers, and various levels of checking to make sure everything fits. Now it's got the same advantage that HOS does; what they have done is set down a set of rules. (HOS has six axioms.) If and when they get that analyzer built, they'll write something in AXES and then the analyzer will pick out errors according to these six criteria. I like very much a lot of the work that has been done with structured design. The problem is that there is no set of rules in structured design that you can tell the machine to check for. HDM has a set of concepts which are enforced by the tools and the specification language SPECIAL. We are using it currently on projects for the military.
- C. McGowan: The HOS approach and others such as SADT show the value of making interfaces visible with automated checking. Interface monitoring is one (facility) that I would like to add as a requirement for a tool supporting the design process. It is a very essential part of design. ... If we think of design as breaking things into parts, the interface is going to be the glue tying them together. You need to make that explicitly visible and controlled, particularly with some automatic support.
- R. Balzer: But I don't understand how that is any different from the self-consistency that you talked about for this particular aspect of consistency. Namely, the kind of message that one guy is going to send is the kind of message the other guy is willing to accept.
- C. McGowan: In DOD oriented software development a normal design deliverable is the Interface Control Document. So you may as well have software support that helps you produce it. Making interfaces visible has high leverage from a management point of view because it allows some kind of parallel development.

- R. Ohlander: Another problem in that area is system integration. ... It is the biggest single problem of development, and it needs more visibility.

[R. Balzer suggests the development of a tool that would look at the overall design and point out areas that are sensitive, so that one would be able to tell which parts are especially critical. C. McGowan suggests some of the possible definitions of sensitivity: high use in terms of interfacing modules, and high impact on performance.

It is pointed out that one difficulty with this is that parts of the design can still be "ambiguous" in that they haven't been addressed yet in the design. Others suggest that measuring performance in general is difficult.]

- J. Machado: Whatever that tool is, it should be able to take any existing module's actual performance statistics into the tool and deliver more accurate overall predictions. When you've got the entire system implemented, the predictions should correlate exactly with its implementation. So the tool in the beginning is an estimator and at the end is actually the evaluator. There should be a one to one correlation.

- N. Finn : ... To really do that you've got to have a model of the system. To know how much some routine costs you have to know how many times it's going to call each of its subroutines and how much each of those is going to cost.

- R. Balzer: ... At every level in the structure, we're going to want to do performance evaluation or estimation. As we get down to lower and lower levels, more and more is defined and so we're relying more on the evaluation part and less on the estimation part. You take a look at any of those tools that are up there and we have the same range of capabilities that they have to operate over. As you are operating in the high levels, there's less of the system defined and so there's more a symbolic reasoning process that it has to go through to figure out what are the range of possibilities for implementations that could exist there. In fact, in general, you're going to find that your tools will be weaker at the higher level because they have to deal with the range of possibilities instead of one very concrete thing that they can analyze. We want the same kinds of information out, we want to be able to deal with the same kinds of tools using the same commands to find things out, and what we find is stronger and stronger versions of these tools as we get down closer to a concrete program. That's why what we're really specifying here is the kinds of tools we expect to exist in this system independent of the level that we happen to be operating at.

- J. Knight: What am I going to do with these tools that you're telling me will be available? We have a basic conceptual problem in a missile. Will this solve all my problems, is that what you're saying?

- C. McGowan: Nothing's going to solve all your problems. If

you're going to raise management issues, you better come up with a schedule and a project plan. You're going to need certain intermediate stages in this process in order to get some reading as to how much progress you're making. These design tools are attempts to help you get some reading, some control as a manager, as to where things stand technically, to what extent is what you're working on likely to be the product that you wanted.

J. Knight: What does that have to do with the DoD common languages?

J. Machado: Let me reply to that, because that's one of the issues that I brought up earlier. The fact is that the language happens to be a component of a larger thing called a systems requirements design development, implementation and life cycle maintenance environment. Language is just a component.

J. Knight: You're going to tell the Air Force how we're going to do our research and development for the next 80 years as opposed to giving us a language. All we want is a language.
(J. Machado : I did not say that)

C. McGowan: You're going to be using the language, and what's going to precede actual coding is some kind of design and probably some other activities as well. Given that we've identified that it's useful to standardize on a language, for many efforts, can we also identify some tools that will help support what goes in front of using the language so that we don't have to keep inventing the tools over and over again. I'm trying to say, what are some characteristics of design as a whole, that are sufficiently carried over to a number of designs so that it is worth automating these tools and shipping them to be used at the discretion of the manager in conjunction with this common language?

[The participants go on to discuss the fact that tools should be integrated into the system, but that they should not be imposed by fiat. It is suggested that the group attempt to catalog tools which are either presently available, or likely to be available soon.]

P. Santoni: For starters, the aspects of traceability and flow consistency interface monitoring, PSL/PSA, that genre of language capabilities. Those tools tend to rely heavily on the human ability to do things like predict tradeoffs. It doesn't have a simulation capability built into it. A similar language and analyzer called RSL/REVS has some of the traceability and self consistency sort of things and includes a simulation capability. HDM, which I spoke of earlier, covers the concepts of how you design a system. It has a hierarchy checker, consistency checker, and interface monitoring. We are working on adding simulation capability to it. There's some work going on at BGS which will eventually develop into a performance estimation and evaluation tool which again will come in. The thing is that the technology exists. The problem is that what you've had is a lot of people researching in the area and not

understanding that they're in a gigantic problem area. Each has tended to take a little piece. Some people who started with a language and said, "gee, we'll worry about how you'll use it later." Some people started with the methodology and didn't worry about languages and computer based tools. There are a lot of people approaching it in a lot of ways. I maintain that the technology is there. The question is getting it integrated.

- R. Balzer: I have another requirement. The requirement is in moving from one of these levels to the next. There is a very big step that is taken and I don't like seeing that step existing only outside the machine. What happens is that the next version arrives and now you're supposed to talk about traceability and proving consistency. What you would like is to have the small steps that were used in arriving at that next level be recorded in the machine as part of the documentation for the development of this system.

[Several of those present agree that a trace of the design decisions would be valuable in its own right, as well as contributing to documentation.]

- C. McGowan: Final point, the Pebbleman mentioned a program design language which I took to mean something that involves the flow control and data declaration capabilities of the source language. I think that would be a nice simple tool that would be easily extracted from our base, the common language.
- P. Santoni: I would like to mention one other requirement. That is that all of these tools, if they're going to be any use to the DOD, better be capable of producing DoD acceptable documentation.

[C. McGowan attempted to summarize the identified requirements. He listed the tracking of design decisions and refinements, design modelling or simulation (which would assist performance estimation), self-consistency checking that hopefully would extend beyond software boundaries to include systems of hardware, software and people, and traceability. Traceability was discussed at length by the participants, and it was concluded that it included both specification of how a requirement is met, and also verification that the specification is correct.

Other desirable capabilities included: budget monitoring in terms of time and space, sensitivity analysis, identification of alternatives, and the ability to assess the impact of a proposed change.

It was emphasized that although many tools were necessary, they should be integrated and computer based.]

- R. Balzer : ... We so far have been talking about the translation from one level in the design to the next. We have not explicitly taken into account the fact that there

may be many people working in parallel on either this level or other levels also making the translation from one level to the next. Certainly, some sort of tool that aids coordination of multiple people working on the same design expansion seems to be quite critical.

- C. McGowan: ... On any large project you're going to want people to work in parallel. Even designers, even analysts. It's by identifying what their method of communicating is, what their interfaces are, that you can allow that (parallel work) to happen and hope to have some control over it. As we get into implementation areas, one of the benefits of top-down strategy is that it does that monitoring in code rather than by management enforcement.
- A. Irvine: We talked earlier about representations for design. The representations themselves are the vehicle of communication. It may be useful to add to our list a separate section, the human factors of those representations. I will suggest that in addition it would be very useful in terms of human factors if there wasn't a great deal of redundancy in the representations of the design process. It's important that people don't have to write the same thing down twice.

[The participants agreed that they were really talking about two things: the use of human factors input to influence the design representation and interacting with the design database, and secondly, the use of computer based tools to assist interaction between people, even allowing for multiple languages, machines, methodologies, etc.]

H. Steubin suggested that this implied a need for a database to store and allow manipulation of the representations. He felt that the database would have to allow for sections of the design that were fixed (as in an interface to a pre-existing piece of equipment) or still in a state of flux.]

- K. Bowles: Add to that the question of how you control the interaction among the people in such a way that you can evolve from one state of the representation to another state without having to go through the trauma of this kind of meeting.
- C. McGowan: Are you hypothesizing sort of a design sequence or process?
- K. Bowles: I guess I'm implying that one wants to make more use of the electronics communication media.

[K. Bowles goes on to say that he desires a computer-based tool that allows for evolutionary development of a design by a group of people.]

- R. Taylor: I'd like to add two more possible requirements. First is the ability to analyze the design, even though the design may still be in an incomplete stage. Second, is to

be able to specify a design which has redundant features.

- E. Taft: you really mean there's a design alternative that you wish to explore.
- R. Taylor: That's only part of it. I'm thinking more of my own evolving conception of the design. I may want to keep some redundancy in it just for my own benefit.

[A. Irvine goes on to say that this redundancy is a temporary state that exists between the time that several similar pieces of the design are seen to be similar until the overall design has advanced to a state that allows them to be drawn together.

R. Balzer conjectures that this type of activity is one of the microscopic changes that need to be tracked by the tools in order to record how a design got to a given state. A. Irvine suggests that such a record is necessary to adequately do maintenance.]

- R. Balzer: ... I've argued that the way you ought to maintain a system is to go into the appropriate level of design specification and make the change and then just carry out the rest of the design implementation all the way down to the coding over again. But -- that doesn't happen. Right now we make the terrible mistake of going into the concrete object, the code which has been highly optimized, at least as much as we had the energy to do, and attempt to make the fix there. ... the right way of doing maintenance is to just make the insertion into this history of design development and then recarry out whatever remains. ... Most of the decisions that you have made before are still going to be the right ones to make and everything sort of flows through. And you wind up with a newly implemented program. And the problem we face is that we don't have the technology to carry out that reimplementation in a reliable way at a cheap enough cost.
- R. Ohlander: You may not have the people. ... Basically, if you look at the type of people maintaining a lot of programs, a lot of the talent (necessary) isn't there. Out there in the real world, you have very low level programmers maintaining so many systems.
- R. Balzer: But there are feedback groups in all of these things in that one of the reasons that you have those kind of people is because of the kind of job it is now. If it required less people to do it, you could afford to put better people on it.

[T. Cheatham expresses the opinion that such a process should incorporate consistency checkers, verifiers, etc. He goes on to say that his experience suggests that such a system makes maintenance enjoyable and tractable.

A. Irvine points out that such a design audit trail will also prevent the reintroduction of errors into the system.]

- E. Taft: I think one of the basic purposes of having languages that allow you to define highly articulated types is precisely so that a certain amount of your design representation can in fact be assimilated by the compiler and I think where we're lacking is that we lack the ability to describe aspects of interfaces and aspects of our designs other than types of data in a higher level design language. But in certain areas we are already able to do that mechanical translation. I just don't really see that we want a clear cut distinction between the design language and the implementation language.

[Its agreed to add a microscopic design report or audit trail to the requirements list being compiled.]

- A. Irvine: I'd like to ask a question about this history. One of the most interesting lines in the Pebbleman is this business about translating design language into programming language. Is it a fact that you can do that mechanically?
- R. Balzer: What Ed brought up just before was that we expect a lot of commonality between those two languages and in fact I'd go a step further in saying that we really should be pointing toward a wide capability language, the subset of which happens to be DoD1 and that language contains lots of non-procedural expressions, forms, that you gradually manipulate out of the language by giving more elaboration of the component pieces. And, you move from one subset of this big language, which is the specification form, to the implementation form, which is also contained in this whole big language, and that's a stepwise refining process, and we eliminate this need for N different languages between the N-1 different steps.

[The discussion continues on the subject of how we get from the top level design level to the DoD1 language. Its suggested that what might happen is that higher levels are processed solely by the designer, and that as one approaches DoD1, more and more automatic tools come into play. R. Balzer expresses the opinion that the design process often has stages in which doing design means taking global requirements and constraints and reducing them into algorithms. He goes on to say that such an activity is one reason that people are necessary.]

- T. Cheatham: An interesting question if one believes that as I happen to believe it completely. is in what notation do you write all that down? And how do you get from that notation to DoD1? Is it a separate language somebody goes off and invents?
- R. Anderson: Maybe that's something we can highlight as an issue to the DOD. This conversation seems like a description of a research project, as opposed to something that might aid system specification in systems that can really be deployed in the real world. I'd like to come back to the point of how much we're specifying here, and whether it will sink the whole ship by adding six times the effort it takes to develop a compiler for DoD1.

[A lengthy debate followed, centering on the issue of how far the recommendations should go and what was reasonable to include in them. One fear expressed was that the recommendations would become fixed and either constrain unreasonably future development, or possibly result in tremendous increases in cost or overhead. Others expressed the opinion that a comprehensive list might raise the sights of system designers, while at the same time, the designers would subset out the ideas that proved unreasonable. The probable use of the recommendations by the DoD establishment was another topic of debate, as was the expected time frame for development of the design system.]

- J. Prescott: One of the comments that was made this morning in the introduction was that the test of this workshop was to get some meat into this document and then the wills and shalls and would likes will be worked out later. ...
- C. McGowan: ... I noticed that Boeing had a position paper on what an environment should be and perhaps you'd like to elaborate on it.
- R. Taylor: I briefly went through it in the requirements session. To summarize, the activities that occur during requirements analysis are similar to those which occur during preliminary design, detailed design, and coding. In particular, the verification activities in each of those phases are essentially the same. It's reasonable to think of a common set of analysis tools which act upon a derivative of the various representations employed.
- C. McGowan: That's an interesting variation on one language; saying that essentially with analysis tools, we want to be able to apply them to different stages during system development. Ergo, we require some internal software formats without necessarily dictating a source language in which it's expressed.
- R. Taylor: Yes, that was the kind of feeling we brought out. At the moment, we can't have one language that is really adequate for all the stages, but the types of things that are done to the various representations are common. Exploit that commonality.
- R. Balzer: And even if the tool is not the same, the user's view of that tool would be the same.
- J. Machado: Yes, the user interface with the system should be consistent. And that should be a requirement.
- H. Steubing: I'd like to suggest that the design language be at such a level that it could be reviewed by, and the final design reviewed by people that are not primarily programmers. [For example] the end user or certain types of management levels.

[This line of inquiry produced a conflict as to whether a unified language was possible or desirable. The

participants agreed that the design work should furnish automatic input to documentation efforts, but were unable to reach consensus as to whether multiple languages were necessary to implement multiple design levels and to allow multiple user communities to interface to the design database.]

- J. Esch: Aren't tools generally interfaced by a database? If we want to write a program in DoD1, how do we reference the information that's in the database? Does DoD1 have that capability now?
- C. McGowan: How desirable would it be to specify or have as a requirement a database interface along the lines of CODASYL?
- R. Balzer: We have to talk about the usage here in that if we assume, as I think we should, that the tools that are going to exist in this environment are written in DoD1, that's where this requirement arises for having the database interface. It's not because you're producing an avionics program that you need this database interface, not necessarily. But if you're going to be writing a tool of the kind we've been talking about here, you do need a database interface. The other thing that we clearly need to have is some representation of programs as a mutable object in the language if we're going to be writing these kinds of tools.

[After some discussion relating to the necessity of a reasonable environment as an adjunct to any programming language, N. Ludham expressed the view that he worried because of the fact that the candidate DoD1 languages were developed without any thought regarding the design environment. It was noted however, that at least some thought is being given to these issues.

C. McGowan called for an assessment of the list of requirements that had been developed. There was disagreement as to the criteria that should be used, but the following were listed as at least meriting consideration:

The tracking of design decisions with support for documentation

A "traceability" matrix

Design modelling and simulation

Sub-consistency of design representations

Performance estimation (trail analysis / budget monitoring)

Interface monitoring

Integration of tools (including standard formats)

The ability to work with partial representations]

Session 3B: Program Documentation
T. Standish, Chair

- T. Standish: I want to begin this session by reminding you of some of the setting. As you know from Col. Whitaker's talk yesterday, during the life cycle of the software that is likely to be programmed in the Common Language, maintenance will oftentimes extend over a period of 15-25 years. The personnel who work on maintenance oftentimes have a large turnover and last on the job an average of perhaps two years. Maintainers naturally have to be able to understand the programs they are maintaining, so documentation plays an important role in their ability to do their job well. Thus, some questions we might try to answer in this session are these: "Is there any technology that can be brought to bear on the issue of documentation for programs? Are there any disciplines of documentation that might make sense in this setting? Is there any machinable form of documentation that any one knows about that could help make the documentation consistent with the program that is documented? I would also like to focus on the sections of the Preliminary Pebbleman Document pertaining to documentation. Do we believe that they are the right requirements? If not, how should they be rewritten? What suggestions can you give? A third category of questions relates to our scientific understanding of the nature of documentation. How, in fact, do people use documentation to communicate about a program and how does documentation help us to understand how a program works? What is known scientifically about the nature of this process? What should comments contain, and so forth?

[Standish goes on to read the sections of the Preliminary Pebbleman Document that pertain to documentation. Section 9.3 says the LSA will develop standards for documentation. Section 9.4 says a master index will be maintained for all documentation pertaining to the Common Language and that formats for all levels of program documentation will be defined.]

- R. Ohlander: I'd like to make a remark. To begin with, first of all there are documentation standards. The Air Force and the Navy already have documentation standards. I'd like to know whether a new standard will have to be developed for the Common Language. If not and if the existing standards are going to continue to exist, we have to address the documentation in terms of those standards.

[Ruven Brooks gave a presentation on the nature of program documentation from his perspective as a cognitive scientist interested in the "psychology of computer programming". Brooks explained some of his theories of how people write documentation and about how they go about understanding programs. According to Brooks, the program text or the program listing is only a surface manifestation of an underlying problem and its solution. The major goal of documentation should be to capture the whole problem solution. Involved in the solution are

facts from a number of different knowledge domains and transitions between the domains. Among these knowledge domains are the original problem domain, modeling domains, where we construct solutions, and the concrete implementation domain of the programming language in which we express the solution. The original documentation should tell people about relevant aspects of the original domains and their properties, it should spell out assumptions made in representing things in one domain by things in another, and it should be explicit about the translations of notions across the domains. Evidence indicates that programming errors can occur in any one of the knowledge domains, and to fix an error a maintainer has to be able to transact in the relevant domain and must apply reasoning in that domain to locate the error. Brooks went on to point out that the only kind of documentation mentioned in the Preliminary Pebbleman Document is the "flow chart", and that while "flow charts" tell about the sequence of operations there is a mounting body of evidence that suggests flow charts are of little help in documenting programs. Brooks cited a study of his that compared the use of flow charts with that of "variable dictionaries" given at the beginning of programs that resulted in subjects being able to understand programs eight percent faster using variable dictionaries than using flow charts.]

- R. Ohlander: What kind of information did you have in your variable dictionary?
- R. Brooks: I should point out that we used very simple programs. In the variable dictionary, for example, if you had an array with a pointer that pointed to the last element, the entries in the variable dictionary would be the name of the array, say, A, together with a description of the purpose of each row, and, if, say, the variable I is a pointer to the last row of the array A, then I would be in the dictionary together with a description that I points to the last row of A.
- R. Kling: Do you think comments should have a certain content which you would constrain by the commenting constructs of the language?
- R. Brooks: Steve Fickas and I have been working on a model of how people go about understanding programs. One thing we notice is that they understand the program in several passes. There may be a top-level structure pass followed by various lower-level refinement passes depending on what the programmer wants to do. If the program contains a lot of comments, with top-level, intermediate level, and low-level comments mixed together, very often finding information is difficult. One suggestion we have is that comments be assigned levels and that you have some kind of automated system that will allow you to look only at the top-level comments, and then to say you want to look at the lower-level comments in a particular section.
- G. Anderson: When you talk about levels of abstraction are you

talking about in-line comments versus a narrative at the beginning of the procedure or subroutines with stars around them which give an overview of the procedures?

- R. Brooks: There is some work by Ben Schneiderman which indicates that prologed comments are superior to in-line comments for small size modules. Let me presume we have small size modules of perhaps four or five pages in length. Perhaps each module has prologed comments at the beginning and interspersed comments throughout. Such modules tend to group themselves into functional subsystems. There ought to be some higher level bigger prologue at the beginning of each of these functional subsystems, and so on up the hierarchy of subsystem levels. In addition, you may need some comments inside each subsystem grouping.

- E. Nelson: I would like to reinforce your emphasis on describing the objective of the program. A great many errors in programs are caused because the user doesn't understand the objective.

- T. Standish: If one accepts your view of the different domains and of the necessity of capturing the translations between them, the prospect is for rather sizable documentation sitting in the background of the program that you say is only the surface manifestation of the problem solution. In the practical environment, unless a documentation discipline is established and enforced, documentation is often skimpy, badly done, and doesn't keep pace with changes in the program. If we accept your view, will you recommend a much larger portion of the overall activity should be devoted to documentation and less to actual coding? What management disciplines would go along with the view that documentation should be more complete in the senses you have outlined?

- R. Brooks: I share Bob Balzer's perspective that rather than creating documentation after the fact as an afterthought, instead it should be created during the program development process itself, and it should involve requirements documents pertaining to the original problem domains and transitions to the modeling domains.

- R. Ohlander: I'd like to agree with that perspective. For large systems, the program documentation is only a small part of the documentation. You also have many requirements and performance specifications along with it. There are design specifications at both high and low levels.

- A. Gargaro: Is the interpretation of what you have presented so far that you are proposing a fundamental concept of the way we design software in our program design languages? We design by levels of abstraction. In order to do this, we are going to have the ability to rigorously define the objects and the sequence of operations. Is that the interpretation of what you may have had in mind?

- R. Brooks: By advocacy of this particular design methodology I am only trying to say what I feel adequate documentation consists of.

- T. Standish: If I can get you to put on your hat as a cognitive scientist for a moment, how would you rate the nature of our understanding of the program understanding process? Scientifically how far have we come? Is the published literature any good, or are we just whistling in the dark in a context in which not much of value is presently known?
- R. Brooks: Not much has been done!
- T. Standish: What could you give as advice for the Pebbleman about current technology that might be applicable?
- R. Brooks: We do know that one thing that has been proposed in the Preliminary Pebbleman Document is of little value. That is that the "flow chart" is a waste of time.
- Col. Whitaker: May I make a comment on that? I know you don't like flowcharting programs, nor do I. I have not been flowcharting for fifteen years. On the other hand, people that are developing large programs today actually do program in flow charts. That is the way it's done. With your experiments, you're talking about a module of 200 lines. When you are dealing with somewhat larger programs, for instance one for which the glossary of objects is 300 pages long, there are differences. Do you have any thoughts on the sequence of these things for such a large system?
- R. Brooks: Let me deal with the flow chart issue first. While there are many different kinds of flow charts, such as HIPO charts, about which nothing is known, we do know about properties of what we might more precisely call the "macro flow chart" --- one which abstracts, say, a hundred line module onto one page. It doesn't show every single operation, and it is known to be relatively helpful. One wonders how big the flowchart at that level would be for a system which has a 300 page glossary of items.
- Col. Whitaker: 5,000 pages.
- R. Brooks: The issue is whether a flowchart of that size is the appropriate way to parcel out the whole structure of the system. As I mentioned earlier, I would suspect that a great deal of the errors and misuse of data objects come precisely from using a flowchart where all you see is the flow of control describing what happens next, rather than using good descriptions of the objects you are working on. In dealing with your question of the 300 page glossary, presumably these can be organized into hierarchical subsystems of objects.
- N. Finn: Can you parcel out objects as you were doing for flowcharts?
- R. Brooks: If you can decompose your whole system into functional subsystems, then for each functional subsystem you ought to be able to have a much smaller collection of data items, and by the time you get down to the module, you ought to have something reasonable in terms of the number of such items you have.

G. Anderson: I'd like to support your contention about flowcharts, speaking from a practical background. I have fifteen maintenance programmers working for me and they all work with listings two feet thick. Not one of them uses flowcharts. While we have flowcharts, those flowcharts are put on a shelf and not one of the fifteen looks at flowcharts when they are digging in and trying to figure out what the program is doing.

[P. Elzer gave a presentation of Nassi-Schneiderman diagrams and of extensions he proposes. The diagrams were applicable to real-time processes, parallel processes, synchronization, exception handling, and so on. Elzer made the point that simple flowchart technology is alone insufficient for these kinds of applications because of restricted control structures, and that for descriptive adequacy in these applications, flow chart components expressing more advanced control concepts, such as parallel processing, were needed. Elzer commented that his augmented Nassi-Schneiderman diagrams provided such a descriptively adequate set of diagram primitives for such advanced applications. Elzer also commented that in his opinion, the use of these diagrams helped programmers to rethink the program structure in a way that clarifies the nature of the problem and encourages devising solutions that really work smoothly.]

J. Meehan: I think we should separate the questions that people ask when they are looking at documentation. This is to elaborate on something Brooks was saying earlier. If you are looking at a piece of code and you want to know "How do we get here?", then maybe flowcharts will help you. But if the instruction you are looking at is " $i = i + 1$ ", none of these things are going to tell you what is going on. Even looking at a declaration that says "I is an integer" isn't going to help you, or even if it says "I points to the leftmost space in this array," that may not help you, depending on what the level of your question is. Even if it said, "This is the last free element of the array, and that's being used to simulate a stack because we are walking through a binary tree which is used to sort some numbers which have just come up, etc.", it may not help. That is, these data structures you are talking about are all simulating something of interest at a higher level. So there have got to be many layers of documentation. Now documents can be very thick, but then the people who come to this program to read the documentation come with different purposes in mind. And it is probably the case that for each style of reader, you have a different style of documentation. If you have somebody who wants a quick overview of what this program does, that is quite different from somebody who has to repair a bug because this machine just burned out or somebody who says he would like to update it to conform to the standards. You can have many styles of documents depending on the questions that you are asking.

T. Standish: Could I follow up on that and ask you --- do you or does anyone else in this room know of any semi-automated

or even automated technology that could help organize this great mass of documentation so a reader could "flip through it", go across levels, or perhaps focus on certain things with "windowing" and "zooming", so to speak?

- J. Machado: I think there is a possibility that a development tool may be a single data base management system or data base which contains all the information about the system development. Another perspective I'd like to put forth is that a good deal of research in data base management systems right now is looking at "views". How do you present to a user the same view of a data base management system even though he may not be looking at the actual DBMS? For example, the actual DBMS may be stored in index sequential form, his view may be relational, and the system may do the necessary transformation for that subset of the data base that particular user is interested in.
- E. Taft: I don't know very much about it, but within the research community, the INTERLISP system has a fairly extensive semi-automatic documentation system that is closely tied in with the programming environment. There are two things: HELPSYS and MASTERSCOPE. HELPSYS is an on-line mechanism that allows you to get at the INTERLISP manual, which weighs about five pounds, and to get at it in a way that relates to the context you are in. So if you have just typed CONS, for instance, and you're not sure what to do next, you could ask a simple question which will be interpreted in the context in which you are writing the program. HELPSYS will reach into the manual and find out the various pieces of information you want to know about that particular construct of the language. Another aid is called MASTERSCOPE, which basically maintains a data base of information about relationships among objects and other things in your program dynamically as you develop and modify it. It's got a fairly sophisticated English like front end so you can ask questions such as "Who calls X in the context Y?" and "How many arguments does procedure X have?". You might ask Teitelman for further details about these systems.
- R. Kling: I have an observation and a question for Col. Whitaker. Some of the analogies we draw are from very rich programming environments such as LISP and INTERLISP. The LISP environments, not the LISP language, really developed in the last 15 years between the west coast A.I. laboratories and BBN. Both of these efforts are very large. They are built of large programs and they entail expensive overhead. They're interesting and they're fun. They've been custom tailored for computer scientists, not for "routine programmers". There is a lot of learning that has gone on in those environments about how to build tools for that user community. One might assume that something analogous might happen in the case of the DOD1 effort. Instead of planning a fixed set of tools that are frozen in the specification of the language, the language environment is left sufficiently open ended so there can be experimentation with different tools for different users. Some of the DOD1 applications could run batch, and on-line documents won't be much help. (And it's not clear what

would be helpful.) Over some period of time different kinds of tools such as documentation aids might evolve. They could be available as utilities or embedded in a later definition of the language. Given the state of the technology now, it's very hard to say what should be delivered in March '79 as a set of software support tools. I assume there is that kind of openness. That is my observation. My question is: "Is there that kind of openness in the DOD environment to evolving tools or is there a sentiment that requirements will be frozen in another two years so they won't have to be slowly developed through dozens of DODI variant systems?"

Col. Whitaker: There is a great deal of discussion needed. Documentation tools are something we clearly need. I'd like to differentiate between documentation, as in Section 9 of the Preliminary Pebbleman Document, which had to do with documentation of the language and the tools of the language, and documentation of programs which is the payoff. Things like INTERLISP and NLS were ARPA developments. My organization built those. But I will have to admit that in the real computing environment they are useless. They are useless for anybody except the fellow who is out there piddling with them. We have been absolutely unsuccessful in trying to transition those to major systems developers. They are not something we can brag about, in spite of the fact that they cost us millions of dollars.

There are obviously some documentation aids. For instance, there are automatic flowcharters. If you like flowcharts you can generate the flowcharts automatically after the fact. Of course, that's not what flow charts are used for. Flow charts are really only used in developing programs and after the fact generation is only paperwork. There are fairly elaborate configuration control systems which include documentation. That is, if you have versions, it keeps track of the documentation of each version and the code for that version, and then assembles the whole system from bits and pieces. That librarian sort of thing has been well used and it is very useful. There are a great deal of internal standards in the various organizations about what documentation to put into the code, identifying the objects in each module and that sort of thing. It is very much individual for each organization and it is very little. I can make up, for instance, a tool which says now I'm going to write a module, now fill out the following things required. Those are useful as a discipline. I'm not sure you call that automatic processing of any sort. There is a move underway to make the documentation you do off to the side in a more formal language format, such that you can generate a program to check and verify that against the program itself. I know of no operational system.

R. Kling: These are all sort of ad-hoc tools that evolve in different environments. Will you provide the DODI environment with many different tools that will be developed on an ad-hoc basis in different places? And are you hoping that maybe over a period of five or six years some core set of technologies will evolve into widespread use?

Col. Whitaker: That's probably a fair statement. The idea of what would be available in the DOD environment is some minimal set of such tools which are, as you say, developed in one place on an ad-hoc basis and could be widely available.

E. Taft: I'd like to make two remarks about Col. Whitaker's earlier remarks. It doesn't seem surprising to me that a lot of the internal documentation aids that have evolved as results of efforts such as INTERLISP or NLS have not made it into the industry at large or into DOD for two reasons. One is that sort of thing just does take a lot more time --- we see that over and over. The other reason is that those systems depend in their very existence on working inside of integrated, interactive environments. As interactive, integrated environments do develop in industry and in the military, I think those tools will begin to see the light of day.

Col. Whitaker: There are a lot of interactive environments and there are a lot of batch environments too, so I'm not sure. Like you say, it's been 15 years. If you don't get in in 15 years, there is some reason to believe there may be more than just the inertia of the system. Another thing disturbs me along that line. NLS, as I say, was developed specifically by ARPA. We have had a resident NLSer who taught all the secretaries to do NLS and there was a time when most of the secretaries were using NLS but none of the project officers. Then we figured out that if none of the project officers were using NLS, why would we impose it on the secretaries? We don't have that young lady teaching the secretaries any more!

[A discussion ensued in which various contributors tried to analyze the reasons for the lack of transfer and appeal of NLS. Then the discussion moved to the general issue of technology transfer. Standish advanced the opinion that while the initial prototype systems, such as INTERLISP, may be too expensive to transfer directly into applications environments, nonetheless, the principles and lessons learned from INTERLISP might well transfer indirectly by influencing the way interactive program development systems are designed in the future. Whitaker advanced the opinion that fifteen years is sufficient time to allow a system to get transferred, if it is ever going to get transferred and that he has seen environments in full scale system operation that are very much better than the INTERLISP environment. Thus, the lack of transfer of such technologies as INTERLISP or NLS may not be due so much to the difficulty of transfer as to the fact that nobody wants that specific technology. Taft pointed out that in the well-developed, integrated programming environments at PARC or SRI, many programmers spend a lot more time at their terminals writing documentation than writing code while dealing with one integrated system. Taft felt there didn't have to be a distinction between a programmer's documentation activities, his programming activities, and his design activities, and that the more leverage a system gives the

less distinction there has to be between those activities. R. Anderson remarked that given a world of limited resources, it might be better not to spend a million dollars on fancy interactive environments when buying 500 interactive terminals for guys who have to wait for listings twice a day might enable them to get their job done with only very simple tools such as text editors. When simple aids are not yet down to the working level there isn't much sense in talking about developing fancy expensive aids for the few. W. Loper noted that he has inherited large systems written by large teams of other people and that he would hate to have to use a simple text editor to search for the next occurrence of a particular variable. He noted that cross-reference tools were very helpful in this context. Loper mentioned a system at Brown University that allowed one to look at the structure of systems dynamically by dynamically picking modules and zooming to get down into their inner details. H. Stuebing suggested looking at how hardware people document and maintain systems to see if any lessons could be learned since they seem to be much more successful than software people.]

- R. Kling: In every session that I've been in so far, and in all the documents I've read about DOD1, there are extensive assumptions made about the characteristics of the programmers that use the language. Will they work in on-line environments with rich computer resources? Or will they work in batch environments with lean resources? Are they going to be people with college degrees in computer science who are highly motivated to keep up with the current technology? Or are they going to be private first classes with high school degrees and six weeks of programming training waiting to get out of the service? Those assumptions about context have never been clearly spelled out anywhere, but they were critical in defining appropriate technologies. We're trying to match a tool to some kind of unspecified social environment. We need a special session to get a clearer understanding of the setting in which DOD1 programming will take place. What kind of personal differences and organizational constraints are likely for which DOD1 should be designed? I think that should receive explicit attention because there is a great deal of implicit controversy about it and presumably by making that understanding explicit, it would be easier to get a coherent idea of what environment would best serve DOD1. I would like to run that session this afternoon as an open forum Session "C".

Session 4B: Program Development Systems
Thomas E. Cheatham, Chair

[Cheatham's opening remarks are summarized in the position paper entitled "Program Development Systems - An Overview". Discussion begins immediately following his remarks.]

- S. DiNitto: I subscribe fully to your proposals, but I want to ask how seriously we can expect that that view will be adopted out in the real world? How successful is NSW going to be? There may be reasons for NSW's failure independent of the issues here. ...
- T. Cheatham: Look, let me try another assumption. The assumption that in this real world, that is the real world two years from now, at least 15% of the programmers starting new projects will have access to a sensible system. If 15% have an easier and better life and are much more productive, say by a factor of 2, that'll make a difference.
- R. Balzer: You've got to start small in the sense you have to prove the things we're proposing are going to really make a difference. If, in fact, you can show in the real world that even with the small user population these kinds of facilities pay off, then good management is going to spread it throughout the military.
- S. Crocker: There's probably something to what you say, but I feel like the experiences have been around for a long time. The systems that you and I use, the systems that Warren showed didn't grow up over night, they go back through a couple of generations of computers and a long history, narrow in terms of how much the community is affected, but long. I think it's worthwhile expanding the discussion a little bit to ask what is it that is necessary to move that technology out. Maybe it's only that there needs to be some controlled tests, that there's some documentation.
- T. Cheatham: Would any of the real world people like to make some comments?
- P. Eastwood: Yes, I think it's safe to say that McDonnell Douglas has access to terminals and large computers. My concern is that we're not part of the ARPANET. If, for example, you develop an editor as part of the higher-order language project the problem is getting us to use your editor rather than our own.
- S. Crocker: That's fair enough and moderately sensible, because we all know that the cost center associated with the computer in our own company has to be supported. There are often more serious issues. That technology will dribble in. The more interesting and tougher environments are the military centers.
- T. Cheatham: Any comment from the military centers?

- G. Anderson: We do program maintenance work, we do not patch binary card decks. We have a timesharing system and work in a higher-order language. We have an interactive editor. We do not have the capability to interactively compile and run interactively, we have to do that batch. I'm from the Marine Corps Tactical Systems Support Activity. We provide the software maintenance for the embedded systems for the Marine Corps.
- T. Cheatham: It seems like you'd be a good example for the kind of place where one could put this new kind of technology. You've gone to the point where that would seem to be the logical next step. Are there some who are so far backwards, who will admit it, and we can talk about that.
- A. Gargaro: I would probably like to substantiate what the last speaker said. I'm currently assigned to the ... Navy program and we in fact do use a program development system to build tactical software. To some extent we have some of the capabilities that were shown in the video tape presentation this morning. A lot of the frills and sophistication we don't have. Basically we do support many of the types of facilities that presentation showed. There was a question this morning about how such a system would be accepted in the military. I can only comment upon the environment that I'm familiar with, and there does seem to be a tendency on the part of program managers that they like to see high paid programmers at their desk with a coding pad rather than at a terminal. There seems to be a reluctance to accept the fact that you can work effectively at a terminal.
- T. Cheatham: So it's time for a management retreat?
- A. Gargaro: Well, that's a possibility, but I suppose we would have to conduct some studies to see if we're justified, if a programmer working interactively is really more productive.
- T. Cheatham: Goodness, that's a cycle everyone went through ten years ago. There was study after study of the effect of interactive computing on programmer productivity. I don't think there's any argument in the world about the distinct increase in productivity with the use of these tools.
- W. Teitelman: I think Dijkstra just took a strong approach on that last year. His approach to writing a program is that you get a piece of paper and you think. [Disagreements] Certainly I'd be the last person in the world to defend that position, but I'm saying there are counterarguments. His answer to why people don't do better that way is because they haven't been trained correctly. It may be a question of education.
- T. Standish: There's a very difficult problem in getting experimental evidence to prove your contention. Sackman, Ericson and Grant did studies a while back trying to measure the difference between the batch and the interactive environments. They did find a 33% shift in the mean in improvement in the interactive direction, but the

experimental designs have always been criticized as not having exactly equal training or features for the programmers and it sets them in a context of factors of 20 differences in the programmer's individual performances, i.e. there's enormous variance in the statistics. It's very hard to nail down and prove that, even though we commonly accept and believe it.

- E. Nelson: I tend to view what you're describing here as various increases in automation of the process, picking out certain parts which can be automated and made to support and called a tool. The extent to which people use such tools will depend on a variety of things, their cost effectiveness, their convenience. In using a system like this, that's a difficult thing to estimate and yet it's very important because there's a certain threshold of convenience above which the average person will work. ... That is, if it isn't at least this convenient, he won't use it.

If you look at it in an evolutionary way you can introduce selectively various tools and various environments. As people begin to see, hey, when you're using that tool suddenly you can do things faster and cheaper and you make fewer mistakes because it's automated and it doesn't make the mistakes that you make on the pad and paper, they'll begin to accept it. I think it's the kind of a system that has to evolve rather than suddenly having a gigantic development.

- T. Cheatham: I would certainly share that point of view and I think one of the questions we ought to try to have a position on is what do we constitute to be a minimal set of tools, maybe the empty set! ... I think one thing, this is not going to happen over night. That seems very clear. I do hear a consensus that although we might not be able to measure the improvement effected by the use of a reasonable system and tools, everyone feels that that is a good way to go.

- J. Bladen: The argument was made earlier that these tools, if they were made available, would possibly not be used by the services because the programmers would not be qualified to use them and would find them cumbersome. If the tools are available, at least in my shop, we can say these tools will be used to the maximum extent. ...

- T. Cheatham: If they're of value and they decrease the cost of programming, then it will show up in the market place no matter how dumb the contractors are.

- S. DiNitto: Possibly a counter argument here. In the Dept. of Defense we've been burned a few times in the past by having a system developed using some useful tools and then not having those tools delivered as part of the final product. So we got a little smart on that and asked for a development system. A worry that a lot of people have is that it seems for each new system, we come across a new development or maintenance system. We've been plagued for years with what we call a proliferation of programming languages. We're

coming to the state now where it may be proliferation of support environments for the system.

T. Cheatham: It seems to me that what we're talking about here is an environment, and I hope that one of the bottom lines of today's session would be some suggestions, specifications, requirements, hopes to lay on this whole DODI effort. ... Are we to the point where we can standardize on a program verifier? Obviously, no. What is the realm of things that could be standardized on at this point in time?

N. Finn: Related to the question of proliferated environments, we're talking about a time scale of 10, 15, even 20 years. Are we putting a restriction here on the idea of diving into the original context of the design in order to make a change? Does that mean that we have to keep the database, the tools that were used in the original design process? Does that mean they have to be around for ten or fifteen years so that we can get into this design then?

R. Balzer: I believe very strongly that the tools that are used during maintenance should be the same ones that are used during development, but that really is tomorrow's session. It seems to me that the challenge that we face here at this meeting and in the development of further documents on the environment for DODI is to find an architecture which we can specify today or in the next short period of time which will support the evolution of tools of the kind that we've just been talking about. We can start with the existing tools that represent the state of the art today and we can upgrade over time to stronger and stronger tools, but yet slip them in an incremental way as they become available. We'll strengthen the environment; more will be known about programs; we'll be able to do better analysis of the things; we'll be able to keep better records of what's going on.

It has to be an evolutionary type process. We cannot hold as standards that which is operational today. That would be foolhardy. ... Our challenge, I think, is building or designing an architecture that's going to be strong enough to support this evolution over time and yet still have the characteristic that systems which are developed early in this framework can be maintained 10 or 15 years later. Now, the tools that they were developed with may have long disappeared, but whatever replaces them has to deal with the kinds of records that were left which I think are very important for the maintenance process. Just getting the source program is no longer going to be acceptable for maintenance operations, even with documentation.

R. Morris: I don't think we have been distinguishing adequately between what kind of tools that are available here and now, thoroughly tested and in wide use; those notions which are blue-sky, under development, good prospects but not yet here; and thirdly those that have been kicked around for a good many years ... It would be somewhat misleading to give the impression that we're talking about a relatively uniform

range of products. We have to, by the end of this session, acknowledge that.

- T. Cheatham: I think we've talked for awhile now about the assumptions. Let me try to shift the ground a little bit and get a discussion going on some of the requirements we may want to imagine being put on an environment for DOD1. ... The question I'd like to address is the question of program representation. ... Text is just not a representation that makes sense for any kind of processing except putting on files and taking off. Would one want to propose a standard representation, internal machine representation for DOD1 programs, modules? Would one want to insist that in the DOD1 language there are built in those datatypes and selectors and so forth that let one deal with this internal representation? After all, the program tools are going to have to. Should that be a standard, is that a reasonable thing to propose? Let me take the strong position of yes, absolutely.
- S. Crocker: If the tools have to be written in DOD1 then I think one is definitely forced into that position. I'm not wholly convinced of the wisdom of burdening DOD1 with the requirement that it be a reasonable language for embedded computer system and that it be able to do program manipulations. I think that those are not closely related to the Ironman requirements. Without debating that point, if you insist we be able to write program development tools in DOD1 then I think it's absolutely essential that datatypes and mechanisms be available inside the language for handling instances of the language. In the words of somebody, programs have to become first class citizens. I think that perverts the language a lot, but I think that's required.
- R. Balzer: It seems to me that there are two issues there that you raised. The first one is that just because programs are data does not mean that you necessarily can execute them. And if you don't execute them, if they're just an object you manipulate, I don't see how it perverts the language at all. It's just like passing around any other datatype, and with that in mind I don't see how there are any extra requirements on the language to treat programs as a datatype. If you can't represent programs as a datatype in DOD1 you probably can't represent any of the other complex structures that were part of the design of the abstract mechanism of DOD1. All that Tom is saying is that we should be able to choose a standard for how DOD1 programs deal with the internal representation of programs. It's really a standards issue more than it is a push in the language.
- W. Teitelman: I just wanted to stress the fact that I feel very strongly that programs have to be able to manipulate other programs, whether this means that the source language is in some manipulatable form or you have access to some internal representation. ...

[Crocker, Teitelman and Balzer continue to discuss the issue of programs as data structures and the desirability

of being able to directly execute such structures. Their discussion is concluded with the following remarks.]

- S. Crocker: You put yourself in quite a bind to say that tools for manipulating programs are going to be written in DOD1.
- R. Balzer: But the alternative is so horrible, Steve. The alternative is that they're written in something else, and then how do you do your transportability? How do you try to maintain some sort of common environment on different systems?
- S. Crocker: I understand that it's just not an easy problem, but I think we ought to have a very clear perception of the size of the problem we're talking about.

[Cheatham suggests that Crocker et al, continue their discussion in a separate technical session. The topic of discussion then shifts to Michener's remarks.]

- J. Michener: The foregoing discussion has been based on an assumption that all the programs in the program database are in the same language. This assumption has not been made explicit, and cannot be fully justified. Some of the tools under discussion will help projects which need to use other computer languages in conjunction with DOD1.

A graphics project might use programs in a graphics language to describe how information is to be displayed and programs in DOD1 to control those displays by supplying specific data values as parameters to the graphics program. (This approach has been adopted for the Advanced Integrated Display System, under development for NADC by Intermetrics and GE.) In such a project, the graphics display unit, being in essence a special-purpose computer, lacks the capability to execute programs written in DOD1. In addition to graphics projects, there will be projects using other kinds of special-purpose computers, like array processors, which will benefit from a similar approach. Of course, there will also be projects with a need for programs written in assembly language.

There should be no reason why projects which need to use languages other than DOD1 cannot benefit from some of the tools we are discussing, like those for history tracing, version control, and text editing.

- T. Cheatham: It would be mandatory. I don't know what has been said that leads people to believe that we're just talking about DOD1 programs.
- A. Evans: It seems to me, Tom, the answer to your question [regarding standard representation] should be yes. The reason is that each of these tools is going to have to go through the same validation procedures the compiler does. If someone writes an interpreter for DOD1, it absolutely must interpret exactly DOD1 and no more and no less, or it's useless. This means the Language Control Board must have the same control over the interpreter and over the verifier

and over essentially every one of those tools that you mentioned, since otherwise there's chaos. The verifier has got to know the same language the compiler does or you're nowhere. The only conclusion I can reach is that they all must be maintained by the same agency and they're all going to have to be standardized.

- R. Glass: I understand the subject of standards, I understand what internal languages are, but I don't understand the relevance of at what level the internal language should be standardized.
- R. Balzer: I think the issue is to take a somewhat larger look at this thing. We're talking about trying to create an integrated environment of tools. To get tools to interact there has to be some standard of communication between them. What we've identified here is one instance, and only one instance, of data which can be shared among several different tools. What Tom pointed out is that a lot of the standard tools we think about - analysis tools, verifiers, compilers - all at some stage or another have to deal with the parsed version of the program. There are many different forms that could be proposed for this standard. It would be very nice if that translation could be done once from the text form into this internal form and thereafter all the tools work on this same internal form of programs. For instance, if the dynamic analyzers work not by augmenting the source text but work by augmenting this internal form and then let compilation proceed to collect the run time data, now you have a more integrated form of operation. What we're trying to talk about is that one of the requirements in this program development system is that certain tools will be required to deposit some of their information in a way that's accessible and useable by other tools.
- R. Glass: There are a lot of levels of internal representation that different tools are going to need. I think you've opened up a bad can of worms.
- S. Crocker: I'd like to kind of agree with both sides of this. I have a fair amount of experience in a couple of projects of trying to share parts of a compiler across the country over the network. It seems clear that first of all it's a very good idea and second of all that there's no way to pursue the standards issue in detail until you've built the language and the pieces of the compiler and until you look at the parts that come out and until you get some experience with that. It seems to me that if one is going to pursue that course one has got to get a first cut picture of what set of tools are a minimal, sort of good working set, get some ideas about what the interactions are so we know what levels we have to tap into. What we've found is that the first cut at what those levels are wasn't exactly right, although it was certainly in the right direction. We've got parse trees out, for example, but we didn't get the symbol table that came with it. We'll build a symbol table from the parse tree, it's all in there, but we would definitely have liked to have a symbol table too. That kind of

experience can't be gained directly.

I don't think the technology is quite here to say "here is the definition of the language and therefore we know immediately what the intermediate levels are". It can be gained by a small amount of additional work about patterning what the set of tools are, doing the first cut design of those tools, and then seeing what the interactions are, and then making some decision about it. I think it's quite reasonable for this group here to recommend a course like that, and unreasonable to recommend anything stronger than that at this point because I don't think there's enough data on the table.

- T. Cheatham: Let's leave aside the problem of internal representation for a moment. Let's see if we can talk about what would be the minimum inventory of tools that would make sense for the initial delivery of DOD1.

[A reasonably disorganized discussion ensue, during which miscellaneous individuals suggest their favorite tools to be included in the inventory.]

- P. Eastwood: I am interested in what we can do to test out timing before we get to the machine. Right now we do a lot of debugging on the target machine by building black box hardware. I'd much prefer to have a software simulator to work with my high level language and simulate inputs, interrupts, etc.

- S. Crocker: We operate such a system.

- P. Eastwood: What is it called?

- S. Crocker: The Prim system. It's a microprogrammable box attached to a timesharing system. We emulate various computers including all of their time sharing system.

- P. Eastwood: And if this were working with DOD1?

- S. Crocker: The current mode is that one writes microprograms that emulate the hardware instructions set for whatever computers you're concerned with. For DOD1, you'd compile code down to the instruction set of the machine that you're concerned with.

- P. Eastwood: So you must generate low level code?

- S. Crocker: You run that under simulated time with input appearing whenever you want. We have complete break point stop capability, so you can debug and repeat any execution sequence.

- A. Evans: We at BBN have had a lot of success with cross-net debugging for small computers. We have defined a simple ARPANET protocol that permits a programmer connected to a large machine (such as TENEX) to interact with a program running in a small one using a DDT-like interface. All symbol tables and source code are maintained on the large

machine. The special code needed in the small machine to support this mode of operation is minimal. ... The concept would be similar to what Steve was talking about, but implemented differently.

- S. Crocker: We can emulate a military computer and we have the kind of scaffolding that sits outside of the emulation that the program normally sees.

[Cheatham suggests that the current discussion be limited to general tools and that emulators for specific hardware are not such. Disorganized discussion then resumes.]

- T. Cheatham: We've had fun playing a game that really can't be played out here. Now certain people want to give their pitches in this session about some existing tools and facilities. Then I think we ought to go back and see if we can in some sensible way indicate our requirements.

[S. Crocker now begins a brief presentation.]

- S. Crocker: This is about a project called AUTOPSY--System for translation of old programs; you can find out how the letters fit into the name. The issue is what to do with the existing programs in the old languages, such as CMS2, TACPOL and JOVIAL. This presentation was originally scheduled as part of the session on tools for migration of old programs. Clearly what you would like to have is some function which would gobble up an old program and generate a new one in DOD1. That's a rather nasty function and we're trying to look at that problem. Our approach is based on some of the points that were made earlier this morning. You have some concrete representation of the old program that you parse, and you get it into some intermediate language form. We plan to translate this intermediate form in some environment. When you successfully translate any program into a form which can be output as DOD1, then you deparse it, prettyprint it and get the new program.

So as to have a reasonable goal, we decided to look at CMS2 as a source. The translation mechanisms will not be totally automatic. There will be ways for the user to apply several kinds of transformations. Also, we feel an important part of the system is to have an audit trail for documentation and undoing. Because CMS2 is not a particularly powerful language with respect to DOD1 and doesn't have parallel processing, we expect to be able to translate most parts automatically. Parts that will not translate automatically will be left as is. We'll provide other means such as the Interlisp editor to edit inline code. Later on we may have general source to source transformation ala in the Irvine catalog. In some cases of substituting code, it may be possible to verify equivalence with the code being replaced. With the automatic translation we would choose some intermediate language carefully and hope that it would have much in common with the new language in DOD1, we would translate as much as we can and go around the hard parts.

There's a research topic in all of this - how to generate

such a system automatically. We have to start with some formal definition of the languages, and construct an intermediate language. We'd have to take that formal definition and look for mappings between the various constructs on the basis of what their semantics are. It's a difficult problem.

Michener: I'd like to speak about program development systems for projects using multiple languages. There is planned a program development system which will keep track of programs written partially in a graphics language and partially in SPL/I which is a standard NAVY language for embedded systems. This system will include mechanisms for associating object code of comparable versions of programs in the two languages. At execution time, there are conceptually two programs: one program, written in SPL/I, executes on a general-purpose computer; the other program, written in the graphics language, executes on the graphics display unit. The goal is to have the general-purpose computer alter the program in the graphics display unit 60 times per second, to achieve visual dynamics. For efficiency, the SPL/I program (in the general-purpose computer) requires low-level details of the structure of the graphics program. To maintain a high-level programmer interface, these details will enter the SPL/I program as SPL/I source statements generated by the graphics compiler. For this reason, a facility for handling multilanguage source programs in an integrated fashion is essential.

[H. Stuebing now gives a slide presentation of the FASP system. A detailed description of the system is given in his position paper.]

- T. Cheatham: I suggest that we should turn our attention to recommendations of this group as to regards to ... the issue of should the tools be written in DOD1 or should they not?
1) They should, 2) they should not, 3) who cares.
- A. Evans: I think they should. There are already requirements that the compiler be written in DOD1. And that requirement was stated before you broke the compiler into several pieces. That means each of those pieces will have to be written in DOD1 and if the language is good enough to do that I can't think of any other tool that won't be equally as good.
- E. Nelson: I wonder if some of the opposition to tools being written in DOD1 is the requirement that 100% of the tools be written in it. There is the fear that some one tool might be awkward or terrible then. If the requirement were not stated in such absolute terms it would be more acceptable.
- T. Cheatham: Let's hear the opposition. Who wants to not write the tools in DOD1?
- H. Stuebing: First of all, DOD1 won't be ready when I want to write the tools. Second, DOD1 is not going to have a data base management system with it and how am I going to write a support environment in that language? Do you want me to do

it in 1990 or now?

- T. Cheatham: Assuming we have a DOD1 compiler, do we want to write the tools in it?
- E. Nelson: Then we're talking about tools built after we have the compiler?
- T. Cheatham: Who says we should not use DOD1?
- H. Stuebing: I've already written my tools at that point and I don't want to rewrite them.
- T. Cheatham: That's a different matter entirely.
- H. Stuebing: That's not going to happen until 1980, or something.
- T. Cheatham: What I want to hear is those who think we should not write the tools in the extant DOD1.

[Scattered remarks and mumblings.]

- T. Cheatham: I guess we're hearing unanimity except for Loveman.

[More mumblings.]

- T. Cheatham: Agreement: DOD1 "shall" write new tools for DOD1. That's assuming it can handle the program as stated. That's the next question.

[Discussion resumes now with the tabled issue of a standardized internal representation.]

- R. Balzer: You could write a compiler in DOD1 which used things like arrays to store the internal representation of data. You could do it in FORTRAN. That's not what we're talking about. We're talking about promoting programs to a real data type in the language.
- A. Evans: It's not necessary that an object program be a class one citizen, as someone remarked. What must be a class-one citizen is a representation of a program, and that is neither more nor less than a collection of bits. It's only when you direct the hardware to interpret it as a program (i.e., to pick it up on the instruction cycle) that it must become a program, and that requires only a transfer function that can be outside the language. You can manipulate the representation all you want.
- E. Nelson: Probably the same point of view here. I think the opposition arises from the implication that any tool has to use the standard, as distinguished from a requirement that there shall be a standard internal representation as a mechanism for communicating between tools.
- T. Cheatham: That's a very good point. Clearly, an optimizing compiler is going to have data coming out the gazzoo about a

program. Let's try to refine the idea. I'm proposing a standard representation in the sense of the way it's talked about, not the way bits are configured on some computer; that's irrelevant. A standard way of notating in NOD1, let's say the result of parsing.

- T. Standish: I'm against the premature standardization of the internal representation in the initial environment requirements. I believe that what we could better do in the initial requirements is to specify the behaviors we would like in the tools and let the representation be chosen by the implementors of the tools to satisfy those external behaviors. For instance, we should write requirements in the form there "shall be break points", "there shall be tracing", "you shall be able to talk to the executive at a break point" or whatever. By specifying the behavior requirements without detailing the representation necessary to satisfy them, we do not overconstrain the possible solutions that may be devised to satisfy the requirements. Perhaps later, when the feasible designs are better understood, we can choose a good standard internal representation, but we do not know enough now to choose a standard one initially.
- R. Balzer: I'd like to expand on Tim's comments because I think they're in the right direction. What we need to do is to identify what kinds of information flow we expect between these tools and for each such information flow there has to be agreement between all of the parties using that information about how they're going to represent things. What I think we really are saying is that in order for there to be interaction between tools, which is one of the goals we set out here, there have to be standards. They may be local in the sense that it's an agreement among the parties that agree to use this data and there may be many such things because there are many kinds of data. And the parse tree, while being a very useful thing for a wide variety of tools, is only one kind of information that these tools will want to exchange.
- N. Finn: The old idea of a compiler is being dispersed over a great many tools now, with a number of code generators, optimizers, editors, deparsers and so on. When the first set of these tools is written, they will have to share a common intermediate representation that will have to be standardized in some form just so you can build on that and not have the intermediate representation shifting under your tools. It will happen but there doesn't have to be just one.
- T. Cheatham: What you're saying is there may be an inactive standard. ... Does that suggest that we say nothing about the subject in our list of things? We understand that people know enough to build compilers in two pieces?
- T. Standish: I believe that we should make a very strong statement that people can react to, so at least the issues will get further discussed. If we state something that's weak and diffused it will probably never be noticed.

Whatever strong thing we say should, in fact, be very restrictive and minimal but have a lot of punch. For instance, all these tools that we speak of should relate to each other and call each other and we should have a decent set of minimal tools. We should make a strong statement of that sort in the requirements so at least if someone else doesn't believe it, they can attack it and thus we will stimulate future discussion.

T. Cheatham: I certainly agree with that statement and I stand with pen in hand to take it down in length as a proposal.

[A loose exchange now ensues between Cheatham, Standish and Balzer wherein they discuss the issues of the intercommunication of tools, the program data base, and other minor topics (not including the weather). This discussion is well summarized in the position paper by Cheatham, et al.

Following a question about the existence of a program library, D. Luckham and T. Standish begin a brief discussion about the contents of such a library. That discussion concludes as follows.]

- D. Luckham: We should be rather specific about the kinds of things that should be in the library. It's my suspicion that there's going to be quite a few years of shake down while people find out how to run things in this language, how to write the modules for an operating system correctly, how to do distributed data base programming. We may as well start with a list of craftsmanship exercises that will go into the PDB [program data base]. I see some of these fancy tools here that people think are very simple for standard languages as not being at all simple for DOD1.
- T. Standish: There may be a point of confusion here. You may want a library of things that have been written in DOD1 available to all users of DOD1.
- D. Luckham: Not only that, I would like to say that people should start working on producing such a library.
- T. Standish: Right, and you may want a user to have to search that before he writes his own program. ...
- D. Luckham: One of the things I see needing to be done first is to build up the library.
- T. Standish: But PEBBLEMAN, I believe, has a requirement that users must consult the program library to see if a module of an exact or similar nature has already been written before they're authorized to write a new one.
- D. Luckham: That's beautiful. I'm suggesting there be something in the library for them to consult. I'm suggesting that we spend two years with teams of experts programming old systems that have never been written in this language.

T. Cheatham: What I think David's proposing is that we need these things, versions of these things, preliminary versions of these things on day 1. They should be started very soon. Is that what you're saying?

D. Luckham: Yes. I'm saying that we better start learning how to control the techniques in this new language.

[Following a few more scattered remarks, Cheatham moves to close the discussion.]

T. Cheatham: What I have on the table, as I understand it now, is a proposed "shall". It's a mumble that we must convert into prose shalls. Let me try a proposition: this body feels that there shall be a program development system with three components, and an inventory of tools. We should have this sufficiently early so we don't have a bunch of people tearing off with old fashioned compilers, but rather people thinking in terms of this new kind of facility.

T. Standish: I'd like to second Cheatham's proposition and then amend it immediately. Have Cheatham appoint a commission of those present, to be picked by Cheatham, to go and write a draft of this over night and ... to have a firmed up version of this by tomorrow morning.

[This proposal was implemented, and Cheatham, et. al. wrote the position paper "Program Development Systems - An Overview" that evening. This paper is found later in this Proceedings.]

Session 5B: Program Maintenance

Chair: Bob Balzer

Summary: Jim Meehan

Maintenance system = development system

Balzer began by presenting the view that program maintenance should be an extension of program development, and that the same tools should be available for both. He offered a replacement for section 7.1 of the Pebbleman document:

DOD-1 will exist within a program development system (PDS) which aids the creation, development, documentation, and maintenance of DOD-1 programs. This PDS will be composed of an integrated set of tools which aid these activities, a data base of information about the programs being developed, and an executive which coordinates these tools and the user's interface to them. The information contained in the data base represents shared information, that is, information produced by some tool and used by one or more of the others. As an example, the parsed version of program produced by the parser is used by code generators, static and dynamic analyzers, and a program editor. This shared information provides the basis for coordination among the tools. As another example, the program at many levels of development is used by the project management and maintenance tools. The design philosophy of the PDS is to avoid the unnecessary re-creation of such shared data by decomposing the large macro tools existing today into smaller component parts which transform one set of sharable information into another.

He went on to describe how the PDS should also provide a symbolic backtrace, in a standard format.

The problems of maintenance

Balzer noted that estimates for the cost of maintenance range from 50% to 90% of the total cost of software. He cited a paper by Lamon and Belotti of IBM, "System Growth Dynamics," which pointed out a conflict between maintenance and program structure: maintenance destroys program structure, and badly structured programs are expensive to maintain. The technology of maintenance requires (1) ample documentation of the system and the history of its development, and (2) programs that produce cross-reference listings and specify the data dependencies in the system.

Maintenance also conflicts with optimization, since optimization requires sharing information in many parts of the system, thus increasing the connections between subparts.

A third conflict is that managers view maintenance as trivial, since their view is from the logical level, whereas the actual implementation of changes is far from trivial. But perhaps the best place to make a change is at the high level of design; the system would then be re-implemented. Of course, that could be a very expensive operation, and we should like to benefit as much as possible from the previous implementations. But that requires that all the implementation decisions be recorded.

Top-down design can aid maintenance

Tom Cheatham indicated that maintenance could be simplified with a good top-down design and the ability to regenerate a system easily. He reviewed the structure of his automatic programming system (explained in a previous session). The descriptions of a program vary across hierarchical levels, but the implementation details do not cross many levels, thus insulating levels from each other. This enhances flexibility, so that if the implementation of a particular data structure changes, for example, the changes need be made at that level and below, but not above. The user must specify how the abstract details at one level are rewritten into more concrete details at the next lower level.

Simple maintenance may require only small changes at one level, and the rest of the system can be regenerated automatically from the same set of rules. More difficult maintenance problems occur when the rewriting rules themselves require modification. Then Cheatham uses a program to say where the rules in question are used, so that he knows where to make changes. He cited an example of an algebraic simplifier where he re-designed the normal form for multiplication, changed 25 programs, and ran verification tests, all in a single day. The ease of maintaining and modifying this system followed from the fact that much of the work (re-designing, changing, etc.)

was done at high levels, and that the parsed version of the program is available as data to all the system tools.

Synthesis precludes maintenance

Colonel Whitaker then described a program synthesizer for certain problems in physics. After the user specified certain input parameters, the synthesizer would create a new FORTRAN program to solve that particular problem, using high-level descriptions of related programs within its library. It also performed all the library-management tasks. Whitaker also mentioned that it had been extended to other problem domains, and that it was far too large to be practical in an embedded system.

What knowledge is needed to maintain software?

The next phase of the discussion centered around the issue of what the maintainer needed to know. Norm Finn argued that it is very difficult to transfer the "web of information" from the designer's head to the maintainer's head via some database of development history. Balzer and Cheatham agreed but said that there was a strong advantage to using notation and concepts from the high-level description, as opposed to the implementation description. Patricia Santoni added that we couldn't do

any worse than we do now, where the typical design document is written for managers, not maintainers, and is therefore misdirected as well as, in some cases, inaccurate.

Dave Fisher raised the issue that in Cheatham's multi-level system, there was the danger that the levels might be incompatible, and that programmers might change the lower levels without changing the higher-level source code. Balzer pointed out that these were both verification issues and might be solved with such tools.

Should maintainers be licensed?

Peter Elzer then asked why we should trust maintenance systems at all, noting that maintainers are generally less skilled than designers, as is the case, for example, with cars. Moreover, the designers and mechanics do not use the same documentation at all. As another example, those who maintain computer hardware go through a great deal of training before they're allowed to work. Why should it be different for software? Sam DiNitto replied that computer systems were often one-of-a-kind, unlike cars, and therefore economics prevented that solution. Tim Standish commented that Germany requires auto mechanics to be licensed, and the U.S. requires pilots to be licensed. He suggested that programming licenses be required for systems programmers

working on Defense embedded system contracts. Captain G. Anderson pointed out that there was a high turnover among programmers, and very few skilled programmers kept a position for two or three years. (Peter principle?)

Big machines or small machines?

The next part of the discussion concerned the relation between the large, comfortable systems with lots of software support, and the small systems for which the code is actually being written. How much of the support software should be on the small machines?

Steve Crocker started the discussion by noting that field maintenance must be as controlled as the original development, and that tactical support out to the field must become part of program development. Colonel Whitaker also stressed the importance of on-site maintenance, as opposed to using some central facility, such as a simulator running on a very large machine, connected by phone lines. But that view, Balzer pointed out, may conflict with the idea that the maintainer must have the "information web" available; instead, perhaps, the software tools could provide only the dependency information, allowing him to home in on the problem.

As the discussion changed focus from long-range solutions to short-range solutions (extensions of current tools), Steve Crocker pointed out the advantage of developing DOD-1 on a large host machine with an ample supply of software assistance, such as the DECsystem-10. Patricia Santoni added that there was a tendency for programmers developing software for a new machine to use the new machine itself, doing self-hosted compiling and debugging, and that if we are really interested in developing a good programming environment for DOD-1, we will have to advocate strongly the use of sophisticated tools. It will require educating many programmers and forcing them to use cross-compilers on the host machine that has all the software tools. Tom Cheatham remarked that the Honeywell Corporation had done exactly that, and with great success. There were comments on the issue of forcing programmers to use the new tools, and it was agreed that DOD-1 compilers should always be accompanied by the PDS so that programmers don't go back to their old ways.

But then Crocker pointed out that such a position implies that any machine that supports a DOD-1 compiler will now be required to support the PDS as well, which could be a severe problem, since the current Pebbleman specifications do not require that, and since there might be resistance to the idea.

Peter Elzer argued against the large-system view, relating the development of the PEARL system which began in 1969 with a large, central facility. The field workers found it unusable on small machines. Communications were a severe problem. Instead, the tools were developed on the small machines.

Ed Taft said that the issue was not whether everyone must switch to some large system, but whether the tools become part of the requirements of DOD-1 "so as to promote transportability." He also added that problems in communication between large and small machines were lessening all the time. Balzer then suggested that there be various levels of support systems, including some scaled-down versions to run on the small machines. The National Software Works (NSW) might provide a model as a central source of tools, although the actual NSW structure is not easily decomposed as the proposed DOD-1 system should be.

Many people agreed that there will be significant financial savings with a PDS, and that the existence of the PDS must become part of the DOD-1 requirements.

Maintenance systems now in use

Balzer asked several people to explain what maintenance tools they use. Captain G. Anderson from Camp Pendleton described his system (MCF-11) in which programmers use context editors, pretty-printers, and very detailed cross-reference listings which include English text, written by the programmers, for each procedure. They are working on augmenting these listings with information about input and output parameters.

Peggy Eastwood (McDonnell Douglas) related that their editor keeps a detailed account of what was edited, who did the editing, when, etc.

Henry Stuebing (Naval Air Development Center) described a maintenance feature in the FASP system (Facility for Automated Software Processing) where the designers wrote conditionally assembled code where they anticipated later changes. They also use a set of test cases after every change; the entire set must be tested before approval is granted.

Several people commented on the need for sophisticated debuggers, with breakpoint facilities, tracing features, and a way to simulate interrupts.

Finally, R. Morris (Bell Labs) related that the maintenance cost of the interstate toll-call switching machines was closer to 99% of the total cost of the system, and that the history of maintenance on these

machines is enormous, far beyond what most of those who favor automated audit trails could imagine.

Session 6B: Test and Measurement
Sam DiNitto, Chair

S. DiNitto:

[DiNitto began by reading Section 6.3 of the draft Pebbleman Document concerning "Test and Debugging Packages". This Section required that test and debugging programs interface with the executive on each target computer and that they have at least three capabilities: (1) selective tracing capabilities at different system levels, (2) symbolic alteration of the contents of memory or other storage devices with memory alteration transactions recorded in a journal, and (3) the capability of printing the history of test events in the journal. DiNitto continued by commenting on some of the realities of testing in military systems. (Unfortunately, the tapes are fragmentary in this Session, and some of the continuity of ideas has been lost)]

In thinking about Section 6.3's basic statement, a capability will be required for diagnosing the performance of programs before and after they are integrated into a system. In military systems, one tends to develop very large test scenarios which are run against the system each time a modification is made. When a bug is found, (and here we've been collecting data for quite a while on cost performance, on a system with a life cycle of about 8 years with regard to maintenance costs), about 15% goes into what we strictly call 'fixing software bugs'. Okay, so now you're talking about 2-10 times the development cost, but instead of fixing errors, obviously not all these errors are introduced during the development stage because of the large amount of development that goes under the guise of maintenance. In looking at 7.6.3 and 6.3 I've come up with a synopsis of basically what they're saying and I think I'll give you some of my own ideas. The first item, 7.6.3, is for a test input and specification control language. Basically, the point that was made is to prevent the programmer from having to generate his own test driver. There are systems which exist to prevent programmers from doing that, but this is the first I've heard of a test input specification language. So, maybe the person who wrote that is here and can shed a little light on it for me. Okay. another point is there should be some sort of symbolic interactive debugger. The next point is mine. From experience we've had with these sort of tools, I think the weapons systems and the defense systems are getting the best benefit from those which do not introduce an overhead into the actual software. There are many debugging systems which do not introduce any overhead to the actual software. Now this means that they have to keep around a lot of information on the programs, e.g. symbol tables, and so on. In addition, one needs other things --- such as dumps, maps, and breakpoints. I was thinking of subtitling this "Yes Virginia, we still need core dumps."

- R. Balzer: I'm having a little trouble with the idea of no overhead in actual programs. If one has a conditional breakpoint in the program, then you have to execute the condition everytime you reach it. How is that, "no overhead" ?
- S. DiNitto: Basically, it's not going to increase the size of the program. The symbol table would be there as well as other information that was needed to find where in the program to put the breakpoints. The programmer would tell where he wanted the breakpoint by symbolic reference, and basically what it would do is to modify one instruction, make it jump to the debugging package, and then jump back when it was done. Dr. Nelson would like to say a few words later about the PACE system. I remember they kept the probes in the code and they amounted to about a 20% increase in the overhead.
- E. Nelson: But usually that isn't kept in the probe when you're running operations unless the people who are running it want to collect some data during actual operation as opposed to during test execution.
- S. DiNitto: I remember there was a particular situation where NASA did leave it in the operational program.
- A. Gargaro: Perhaps if I can just relate what is done in the AEGIS program using CMS-2.
- [Gargaro proceeds to describe the debugging tools in AEGIS which are summarized as follows: First there is source debugging. Second there is an interactive debugging system which works off a symbol table generated as part of the object code. Finally, a third mechanism is a tool which can go through the source code and insert more specific types of probes].
- S. DiNitto: Contrary to what I heard this morning, patching is still done. At Tinker Air Force Base, they have a goal that after experiencing a problem, they're supposed to be back up in 20 minutes. What they do (and it was one of the strongest reasons for not using high order language), is to look at the problem in binary, or octal, or hex, --- and to come up with a solution. Then certify it, put the patches in, document it, and get the system up. Now they only reassemble and relink once every six months which makes it a real problem for using higher order language, because you cannot always guarantee you will get the same actual instructions out of the compiler that you would from the assembler.
- R. Balzer: It would seem to me that one could have this capability by using an incremental assembler which, given the symbol table from the output of the original assembly, would allow you to use source level assembly statements and would do the patching for you. That could be done with more accuracy and reliability than people could do it. So you could have a tool which helps you operate that way and it

could also do your audit tracing.

S. DiNitto: Right. I'm hoping I can spur some ideas on how we do it in an HOL. Another thing in the PEBBLEMAN was the requirement for some sort of hand tester or automated verification system. TRW has one called PACE, GRC has one they call RXBP for FORTRAN. We've got one called JANS (Jovial Automated Verification System). Basically they provide the ability to trace, and to provide a frequency of statement execution. At least in JANS and RXBP, we've been able to make it do the same thing. Another item is development of efficient test scenarios. They do provide, in some of the systems, certain assistance in developing the test cases. You need names of a particular module, unit, or statement and information on "how do I get there?" Now mathematically it can be proven that you can't do it in every case. But we've seen, in the majority of cases, that it can give you some nice hints. The other thing is timing information. How long does it take something to execute or what's a good estimate of it? As the compiler goes through the generation of instructions, it has stored away a table of times for particular instructions. Now it can't be a hundred percent accurate, but it can give ideas of the size and the timing. We're talking about tools and I think the testing process takes on the average 40% of the development time. Yet the PEBBLEMAN doesn't seem to address those tools very well, in my opinion. Dr. Nelson has agreed to give us a little rundown of their experiences with a couple of systems.

E. Nelson: We started back in 1969, being concerned with the questions --- "How much testing is enough?" and "Is some of it redundant? Next could we develop some objective measures of it? One of the first ideas was to determine which statements are executed in a program. An initial program was developed called FLOW which could instrument a program and count the number of times each statement was executed. Soon they discovered that in large programs they were collecting a lot of data and that created problems. But one could analyze the program into structural units called segments, which are such that if you execute the first statement of it, all the rest of the statements in the segment are executed. The remaining statements would be the transfers or the branches between the segments. So we developed a system called PACE, addressing this whole question of the collection of data, building test data bases, and using this information to develop a measure of test effectiveness which was a very simple fraction of those segments that were executed to find out how we could generate test cases that would exercise all of these statements. We developed a better algorithm to collect data and the one Sam mentioned for NODAL is a quite cost-effective one, in that its expansion and core is very small and on execution it's very small --- in many cases under 5% and rarely over 10%. One of the things we did was to take some programs that had been tested by conventional methods and run them through the test analyzers using original test cases. Then we took the faults that were found in the operational situation and analyzed where they

occurred. Conventionally prepared test cases did not exercise all segments, and almost all the faults found in operation were in those segments not exercised. This stimulated interest in the tools and their use. One of the areas we've had quite spectacular success with is in delivering operational targeting programs. In about 1973 when the first delivery was made, a rather spectacular reduction occurred in the problems encountered in usage. In this first one delivered, in their whole acceptance testing they encountered no problems and over a lifetime of somewhat over a year, only two small problems were encountered. To understand why the tools work as well as they do, one has to look at testing. You're testing against the functional requirement. You make a model of what the program is trying to represent and verify that the program corresponds to this model at various points and develop an inference as to its behavior at the points in between. So test tools of this type can be of great value to catch a great many of the errors, and to produce dramatic increases in reliability.

- R. Balzer: Two questions: On that study that you mentioned, most of the errors occurred in the untested portions of the program. What percentage of the errors were found in that untested portion?
- E. Nelson: Well, the sample wasn't necessarily very large but I think it was over 90%. I wouldn't want to draw a general conclusion from it.
- R. Balzer: The second question is, what percentage of your development cost do you typically spend testing?
- E. Nelson: Somewhere between 40% and 60% of it on the large systems.
- R. Balzer: That includes debugging them, too? I.e. getting rid of the errors you find?
- E. Nelson: Well, one of the faults I would find with the [Draft PEBBLEMAN] document is that it treats test and debugging as all one subject. We tend to break it up into unit testing, integration testing, and finally system testing.. There are different types of tools used in each phase. In system testing, one of the important things is how does the test case relate to the requirements. So one has to develop a mapping of the test cases back to the requirements and to particular sections of the code. There's another aspect of these tools. Finding an error and fixing it means you've changed certain of the structural components, and there are two problems here. One is that when fixing an error you can introduce errors too. One of the things we found by having structural information telling you which sections of the code were executed is the criterion that you've fixed the error is "correct execution of the test case that found the error". We say you should also cause execution of another test case which exercises different logic paths so it will have some different means of entry into these parts that were corrected and relates different sections of the code. In maintenance, you're changing only a part of the code. In

the retest, you want to choose only those test cases that will exercise the parts that were changed. So you can use structural information to reduce the amount of test cases.

- S. DiNitto: I'd like to pose to you one other item. On the subject of identifying phantom paths, I was wondering if you could get into that a little bit.
- E. Nelson: Applying our interpretation of a program as the specification of a computable function, you can partition the input domain into disjoint subsets which exercise different logic paths. Then the question is, how many of these paths are there and how are they related to structure. If you construct the paths from a branching table many of these paths are not executable. I call them phantom paths. In small programs, there may not be too many, but the larger they get the more they have and the paths that are executable are a small fraction of the total number of apparent paths. Also, phantom paths complicate the writing and checking out of programs. If, say, only a few percent of your paths are executable, when you look at the code text you have no visible cues to tell you what the real logic structure is. So the only way the programmer can find the errors is when he goes to execute the program. So one of the questions raised was to identify which are phantom and which are executable paths, and, having done that, can we rewrite the program so there are no phantom paths? I also found another source of artificial complexity. In constructing the actual functions that a program performs, we found some of the paths computed the same function, although they went through different paths and so there was a functional redundancy. Also another thing was found, this being particularly true in those cases which were data processing oriented as opposed to being mathematically oriented. People were used to thinking about functions in the mathematical sense, but when you come to a data processing problem, they don't usually think of that as a function, but it can be described as a function too. We also found that on some of these programs, when they were rewritten they not only had fewer higher order language statements but also compiled into less object code and took less execution time.
- S. Gerhart: How would various control structures or various notions of structured programming influence the phantom paths? Would their use introduce more or less?
- E. Nelson: We found that most of the programs when rewritten in this functional form were also structured. In many cases they were initially structured. Typically we found that the average programmer in applying structured programming follows the rules artificially and this contributed to making phantom paths.
- S. Gerhart: It's easier to put in a boolean variable, isn't it?
- E. Nelson: Right, or to put in a flag that keeps you reminded that the decision has already been made. There were some cases in which the structured form did not have the simplest

structure, but in many cases, when you do eliminate the phantom paths and the functional redundancies, the result usually meets the criteria for a structured program.

- R. Glass: Are phantom paths determined based on the coverage?
- E. Nelson: Well, to identify a phantom path you have to show that there is no input that will cause it to execute and what we did was construct the input domain and construct the various subsets that belong to each executable path and then find out whether you covered the entire domain.
- R. Balzer: First, how much of this phantom path analysis and restructuring is automated and how much is manual?
- E. Nelson: At present it is manual, but a lot of it can be automated. Now some of the data for this actually came out as the by-product of tools like NODAL so that there's some partial automation.
- R. Balzer: The second question is about the restructuring. Is it basically a technique of copying code in a place after a join and a forkout. Is the idea to copy code and build simpler paths?
- E. Nelson: Well, I developed a notation in which you could represent these various code segments and branch expressions. Then using that notation one builds the functions that it performs. Having set up the functions in this notation, one then looks for common elements in them and develops a branching expression that covers the common elements.
- R. Glass: There are a couple of categories of errors that you don't get at by the totally rigorous use of a test coverage analyzer. One of them is pieces of logic that have been omitted when requirements aren't satisfied in a program. Another is the more difficult problem of combinations of logic paths, where a program may fail because you went through that logic path after going through three other logic paths to get there, and only that combination would fail.
- A. Gargaro: I reviewed a paper which dealt with the formal testing of software reliability. The methodology that was proposed was that you determine an input space using Dijkstra's predicate transformer. Basically, what the paper tried to do is extend the concept to determine software reliability based on a set of weakest pre-conditions going into a function to be satisfied. Normalized weights are assigned to successful outcomes of these pre-conditions being met. The paper goes on to suggest that once you have this measure of reliability, you can perform empirical testing and derive Bayesian statistics to determine when the software reaches the desired reliability. I should caution that in this paper there were some extensions to Dijkstra's work that I did not think were justified.
- E. Nelson: Note that the subsets that I spoke of here are a

more simple terminology for saying what Dijkstra calls 'weakest precondition'. It's a specification of the subset that will cause execution.

S. DiNitto: Thank you Dr. Nelson. I think it's obvious path testing isn't the total answer but one of the best things we've right now. Yesterday in Dave Luckham's session, we talked about formal verification and it's quite obvious that isn't here right now, at least for a system the size we're talking about. I think path testing is really state of the art right now.

S. Gerhart: We might mention symbolic executors, which are between the path testers and provers. In the limit, it would reach a proving kind of situation but these have been found to be useful although they're not easy to use because they require some theorem proving capability, but they are a little more effective than testing with actual data.

R. Balzer: The very first FORTRAN compiler had as part of its compilation process, a symbolic executor that determined which part of an if statement was more likely to be executed first and used that as a basis for deciding which branch to put immediately after the conditional and which branch to reach by a skip statement. So this symbolic execution technology has been around for quite a while.

S. DiNitto: Also, that first FORTRAN compiler made a lot of phantom paths. I was told there was something on the order of 20,000 instructions you could never get to. That might have only been one path. Do we have any other examples of interactive debuggers, path testers, etc.?

R. Taylor: I think we should think of testing as not only occurring on the end product, but also on higher level representations of the program, because if you are able to symbolically execute your design or, in some sense, test your design, you're going to be more sure of your production process by way of the fact that you generated a great deal of feedback.

S. DiNitto: I think that some of that is done right now, if you follow IBM Chief Programmer Team methods, using walk-through at the higher levels. As far as automating that, I don't think they've done that much work.

E. Nelson: There has been some effort in what we call functional simulation. It is taking a preliminary design and developing a simulator at that level. Simulators may not be a hundred percent representation [independent?], but wringing out a design with a simulator often is better than waiting until implementation.

A. Gargaro: There's a certain class of errors that I don't think is ever going to be detected except through very exhaustive testing. For example, the compiler interacting with its run-time system. Throughout this Workshop I've had questions pertaining to run-time systems and to just how you

isolate them from a compiler or compiler tool. We have a compiler we've tested exhaustively, and we are very confident as to its reliability. We have developed a very high degree of confidence in the compiler's reliability, and we may have done this through giving it a number of test cases, or it may have been put through a verification system. Someone in the operating environment makes a change which impacts the run-time system the compiler interfaces with. For instance, someone enters another error code in the run-time system. This error code is very important and should be picked up and analyzed by the compiler. However, the compiler writers were not notified of this, and when the compiler executes it gets this value which it knows nothing about, and the compiler fails. I don't see any way around determining this type of failure except through testing.

- R. Balzer: This is a clear case of maintenance. The environment of this program has changed. That changes the specs. We've argued about how one does maintenance as an extension of the development cycle. You must have that either the change doesn't affect this program or you must find those places that it does affect. Once it is identified that the environment has changed, and that it's changed this particular program, then it is just like any other maintenance problem.
- S. DiNitto: Why are we so worried about testing? Why not just shake it down in operation for a while? In some cases, it's obviously not possible, and in some cases it isn't really that serious. It's just the general cost --- an error costs about ten times more to correct if it's detected during integration testing. In the maintenance phase, the cost is up around one hundred times what it would cost to fix it during the test phase.

-----: I just want to say something about the testing environment. Usually there are two separate environments and I think they both need to be dealt with. The first is a very favorable one that typically takes place on a large host computer with a lot of tools available. Another environment is the real-time computer where it's eventually run, and that's usually a very sparse environment with almost no tools available. I think that's the environment where we need the most help.

- S. DiNitto: I'll have to agree with that. We've come into contact with quite a few situations like that where the real computer's even in the development system. One thing Jim Prescott and I discussed briefly yesterday is "what is the test specification language? When we had talked about it we were thinking about it is a way to control production of tests, and the modification of tests. I think it's quite obvious that in a lot of cases the tests become much larger than the actual operational software. Have you thought anymore about that?

- J. Prescott: We would like a test driver of some sort that gives the programmer a way of specifying the domain of inputs for his program rather than having to write them all

out by hand --- some sort of a test specification language to tell a test driver that these are the inputs I want delivered to my program, these are the sequences that I want certain pieces of the program executed in, and these are the outputs that I would expect out. That language would check to see that what does come out is verified against what is specified should come out. Concerning the topic of quality assurance, personally I've never been too impressed with the test data that have come out of this quality assurance effort and if I can extrapolate it to a general weapons system I have a feeling that the same situation exists because in many cases it's the same people developing the software. Bob, can you shed any light on this? How is quality assurance organized right now? Or is this a recognized deficiency? Is it a brute force thing?

- R. Glass: The brutally honest answer to the question is software quality assurance, as I see it applied at this point in time, is typically a rather control oriented, measure-the-obvious, kind of thing. We do a lot of things about configuration management and library control systems and document review, but we really don't do very much about the quality of the software itself. I think there is a historical reason for that --- most of the people in software quality assurance, in my company at least, came from the quality assurance side and don't understand software very well. Did I answer your question?
- J. Prescott: Yes. I think a lot of the tools we've talked about here should be tools for those SQA people to come in and determine the amount of coverage that the testing has done in the software. So I think that the people who should be looking towards using a lot of these tools should be the SQA people themselves.
- R. Taylor: I think it's important that those tools be used in the proper manner though. You can have a quality assurance organization that operates basically in an adversary role. TRW has a program called STRUCT which does a code audit just checking to see what kind of construct a person used when they wrote the code and the environment where it's used. Here's a tool out there to help you keep an eye on what you're doing.
- R. Glass: I also want to say some positive things. It's also the responsibility of SQA in our company to do some of the things that Taylor talked about in providing the tools for the project world. In fact, one of the tools we're building is the test analyzer --- a rather weak version of what Dr. Nelson talked about earlier in this session.

Session 1C: Training and Education
Kenneth Bowles, Chair

- K. Bowles: If we want to have a provocative discussion I think we should have one of the service people give us some guidance or we will be talking about two worlds. My feeling would be that for the language implementation that's going to be widely used starting in the 1980's, training is going to have to be developed well before the language is available to use -- a chicken or egg proposition. Do we start predicting within the restraints we know about what the language is going to look like, work on some of the problems, and train people to work with the language at this time; or wait until the specifications are known in detail and then start constructing teaching materials which will require a year or more to put together?
- P. Cohen: You talk about mainly training the programmer and possibly the first and second level supervisor of the programmer. Note that the DOD language will be used for embedded computer systems, and not for a software project where the software manager is the program manager. In the systems we're talking about the software manager is usually the manager of one portion of the project and there is a whole systems organization over him. In order for the language to be accepted, the overall project manager has to be aware of the software technology that we're trying to adopt.
- T. Standish: Dave Fisher told me that he had a question he'd like put on the agenda -- How can DOD encourage universities to help with the training of people competent in the language and the environment in such a way that it benefits DOD?
- J. Shen: A lot of times, even though we know the language or the material we have a hard time trying to teach it to people. We should be aware of the levels at which we are teaching -- we are going out to teach service people on the language, but first we have to teach the instructors to bring the message to the service people.
- K. Bowles: In San Diego, by and large, I have the impression that the operation is focused on training new recruits -- people that are low scale compared to what we're concerned with here.
- P. Santoni: You'll find most people have little background. They come up through the Navy. You'll find the minority of people who are programming today are from the universities. Very few come straight out and go to work on programming systems. Most of the people you find learned computer science from the ground up. They were mathematicians or physicists when computers started to be used, and they became programming staff. Their background may be from COBOL to FORTRAN to maybe CMS2, but mostly assembly languages.

J. Shen: [Mr. Shen pointed out the necessity for understandable and interesting reading material in presenting and teaching the DOD programming language.]

Col. Whitaker: It's true that we do not have good manuals for any of the DOD languages. That is very important. It's not all that good on the outside either, but [in DoD] it is so shockingly bad -- there just isn't anything at all. A good manual is vital. The most useful thing I find in any manual is examples. That's something they seem to avoid at all costs.

S. Fickas: Why are they so bad? Is it the state of the art? Can't people write good manuals?

Col. Whitaker: Writing a good manual is a very difficult thing --- not something that you just knock off, or give to a contractor as the lowest bidder. ... For managers there would be an interesting point in adding 16 hours to the Defense Management School. The School is an organization which trains systems program directors. You go from there to manage a major program. They have a software engineering thing now.

S. Fickas: Where is that school?

Col. Whitaker: Fort Belvoir. It is a Defense wide thing and that is the key. There is another managerial operation called the DOD Computer Institute at the Washington Navy Yard and you would think that it teaches all about computers to everybody in the DOD. But there is not one person in a hundred in the DOD that knows it exists. They offer 50 or 60 courses, but I've never heard of a graduate. For some reason they seem to be very obscure. That is a place where you could develop courses, and they're professional course teachers. They can even develop courses and go out and teach them at other installations. The Air Force has a school for instructors. The Air Training Command teaches programmers and the Navy has a similar operation. The ones that go through that sort of exercise really do get very good training. Most of the military development takes place during contracts. Civilian contractors get somebody off the street, and he gets 5 days of training, and he is a programmer. He is released to play with your bits. It's not just DOD --- that's how your airline scheduling program got written too -- the same company wrote that too.

R. Kling: I'd like to address the quality of language manuals in the military. There are good electronics manuals I've seen coming out of the Navy. There's a difference in text in manuals. In the private sector, the best computing books are language texts. Application manuals are written by the manufacturer, but FORTRAN and LISP books are written for a textbook market and they include examples that really guide people through the language.

K. Bowles: How do you see the impetus coming in the way of programs or contracts or management decisions that will put resources into causing some of these things to happen?

Col. Whitaker: It's very difficult to say. We can initiate some materials and techniques in this area, indicate that there is a large enough market, and encourage other places in the DOD or outside to do it. The training has to be done by organizations that do training today. The key is to show that the training works out.

J. Meehan: I think there is a distinction between manuals and texts. Manuals are written by programmers and tend to be very technical and are horrible for learning. That's where all the answers are. If you need to know how some function is implemented that's where it will be. Texts are generally written years later by people who've had a long period of experience in teaching the language or using the language. The talents for writing manuals and writing texts may not be at all correlated.

D. Kibler: I think it's unfortunate that we don't have someone here from IBM because that firm has a great commitment to education.

T. Standish: IBM was responsible for programming the NAS Stage A Air Traffic Control System which was initially tried out in Jacksonville, Florida. They scraped up ninety day wonders with ninety days experience in PL/I, and set them to programming this immensely complicated thing with a football field full of radar scopes, complicated flight strip printing, graphic data block handling with a lot of complicated, mechanized hand-offs, and an interface to the controllers. With absolutely appalling cost overruns they had to replace the computer from a single 9020 to a double, to a quadruple, to a Model 65, to four Model 65's and then ultimately to 67's because they mis-estimated the amount of code. The first software release was absolutely a marvel -- 426 controllers signed a petition saying "Get it out of here!" So it's obvious that the industry, even if it does have a focus on training, has in some cases done appallingly poorly in making sure people are certified and competent before they are hired onto these enormous system jobs. And so, perhaps, taking up on something that Whitaker said, there should be something like a certificate or a programming license. Just as you don't let people fly airplanes without certification of some sort, maybe you shouldn't let them touch a computer until you get your programming license or your DOD rating or something like that. I guess it isn't our role to come up with management solutions here. Rather we're supposed to be addressing technical issues. Nonetheless, you could write contracts with teeth in them by saying to a civilian contractor that "If you're going to bid on this, you have to certify that each one of your programmers has a DOD1 rating."

J. Shen: I feel another thing we want to make sure of is that DOD right now is a transitory service where people come in and then leave. And if we do not make training much easier to learn, the guy who takes two years to train -- by the time he's done, well, he quits because he gets hired by industry and gets higher pay.

R. Kling: It's a misunderstanding of technology to assume that it's easy to focus on technical issues and then to assume that managerial issues can be dealt with easily by someone else. I've got a position paper which deals head-on with that. To assume that a language like DOD1 can have positive effects under random managerial concept control, which are not specified as necessary parts of the language use, is really a fundamental error.

K. Bowles: I would suggest briefly one approach to training that I think may be one possible way of working with a population of people who are at least as motivated as beginning college students. I'm talking about use of the training method which is the personalized system of instruction. It is a method which recognizes that conventional lectures are a very poor way of communicating to the students. It's been proven in a recent study of one large population of students that the lectures contribute virtually nothing, particularly in this environment of self-paced, personalized systems of instruction. I feel one should think carefully before making a big expenditure -- comparing the expense to implement automated quizzes, for example, or drill and practice material that leads up to a student being able to evaluate how fast he is going through the materials, with the instructional staff to evaluate how fast they are going through.

[Bowles points out the value of having students progress at their own pace.]

I think that since we are talking about a fairly large number of people, that with an investment of in the neighborhood of 50 hours of preparation time per contact hour, something should lead to a number of different programs being well within the reach of investment. We also use people we call proctors. In the university environment, these are junior or senior level students who have taken the same lower division course as the freshman or sophomore. You can make a modest investment in automated practice and quizzes, and increase the rate of productivity of these proctors by a factor of 2 or 3. We could probably do better than that by further working on the text and course materials to refine some of the areas which we discover are sticking points for a large number of students. But in the meantime, we have succeeded in largely eliminating the routine duties of the proctors.

T. Standish: One of the ideas that strikes me as very valuable is to have examples of large complex programs published in the literature of DOD1 for study and learning. In England, Tony Hoare is trying to start a series of publications where you take large pieces of software, such as operating systems or text editors, and publish the entire listing with a lot of comments. Whitaker said this morning that there's a factor of two improvement that can be achieved just by better training at one level. There's also one other well known phenomenon --- the Software Learning Curve. It says the second time you do a similar system to one you've done before, the resources needed are cut in half, and sometimes

the third time around they go down to one sixth. Your estimates of how much you're going to need get better and better the more often you do it. There is a tremendous benefit not only with being familiar with examples of the language as a medium to write programs, but being familiar with the kind of programs you're going to be writing.

P. Cohen: I feel we have to take a transition plan approach to training here. We have our programmers now who are used to using assembly language, FORTRAN, etc. They have to be trained in using a new high order language.

(unidentified): Our experience this year with trained system programmers who just graduated with experience in the use of Pascal, is that they are in such great demand that it's hard to believe. They've had the pick of the large industrial firms. It's an indication of the sort of demand and the fact that it's not being satisfied.

P. Cohen: I don't believe the same training in Pascal given a FORTRAN programmer should be given to a machine language programmer. The FORTRAN programmer should first become familiar with the cognates in the new high order language of FORTRAN statements, and go from there.

J. Shen: I'm still trying to emphasize that DOD environment is trained by the people we have are not highly sophisticated. Most are high school educated, with average IQ of around 80 or above. We have a high turn over problem. When the DOD high order language becomes available, one solution I'd like to see is mass propaganda, like the communists have done.

Col. Whitaker: Advertising we can do; propaganda I'm not sure is proper. Perhaps the more important thing is that your system is built on the machine. In teaching you to use the computer, there is the strong connection there -- you can play with it. It's like teaching anatomy and not having a cadaver. There is much stronger reinforcement and I would expect much stronger productivity in training benefits when teaching computer things.

D. Kibler: I think it's a mistake to think you're going to create a language that's going to be usable by every high school graduate. There are several indications that that is the wrong way to go about it. Dijkstra remarks that the reason software costs so much is that software firms hire cheap programmers. He means that the programmers are not well trained in the language or in the knowledge domain they happen to be working in. I think this is indicated further by the Chief Programmer Teams that have been used by IBM.

K. Bowles: I keep hearing "high turn over", "high school background", "not motivated to stay in the service", and so on.

Col. Whitaker: I wouldn't speak about the Navy, but that may be a little too grim a picture of Air Force programmers. The best programmers I have ever seen in my life were airmen. Not to say all the airmen are good, but they include the

best I have seen. They commonly have a year or two of college, or are high school graduates.

K. Bowles: Which population do you really want to have brought up to understand this language in the near term?

Col. Whitaker: Unfortunately the ones that will be using in '80, '81, and '82 will be primarily contractor programmers -- CDC, SDC, IBM -- because they will be developing the systems. We don't have control over that.

R. Kling: Would that change by the late '80s?

Col. Whitaker: Programs go into maintenance and the military often takes over.

R. Kling: Does that mean that the kind of programmers that Shen's talking about will be more involved? Programming concepts have to be intelligible to a wider range of people than just the first developers.

P. Santoni: What they say is very true. With contractors you'll find that they have a development facility. When you talk about an initial prototype of a system, you'll find maybe one or two civil service people, rarely any military people, perhaps developing some code, but mostly monitoring a contract. When it comes to people who sit down and write the military systems, very few of them are fresh out of the university. Most people you find there have been through junior college computer science. At the outside they have a couple of years of any kind of higher level education, not necessarily in computer science, but may be hired on as a programmer, and their computer science backgrounds are totally different. Their application experience is going to be small. Those people have no background in the actual system they will be using, and there is going to be a long training process. Right now the training class in CMS2 runs a month.

[Santoni goes on to point out the practical difficulties of getting managers to schedule training time for their people.]

K. Bowles: Would it be possible for folks like that to be reassigned for, say, 25% of their time over a certain period to work with package course materials.

P. Santoni: It would be beautiful, and some of that, in a very limited amount, goes on in my group. ... One of the things we have tried to do in the project I'm working on is to educate the management people to the overview. The "What's is going to do for me?" -- not the restrictions. One way we've done this with the PSL/PSA system is to put together about a 3 hour video tape lecture aimed at the management level, telling them what it will do for them. A very important part of this is to get people motivated to put their people in the training or to allow them that 25% of their time.

- P. Cohen: I'd like to pick up Shen's point again about the propaganda. I think we do have an opportunity to propagandize this language. We will need cooperation of such organizations as the ACM and the IEEE Computer Society. We should get someone to give us a half page in a journal each month for a column for what's going on with the DOD common high order language effort. That certainly is a legitimate way of advertizing.
- P. Santoni: Again, you've got the people in the trenches who are managing the people who are writing the code, and you've got to convince them somehow to loosen up and free up their people for on-going education. Some of them are very open-minded, and some of them say "I've got my people stacked just like that. You want an assembly language programmer, you want a programmer in CMS2? I haven't got time to train him; what we'll do is circumvent the CMS2 requirement."
- J. Shen: My boss would sent me to a programming course and when we come back we can't project six months and so he would say "Forget about the structured programming. What we did in the past we got by with right on schedule; you can try it on your own." How many programmers are really doing structured programming? Many people at the universities, maybe IBM. We have short courses, week courses in structured programming, design, it's all for designers, how many courses are offered to managers? One day, half day on structured design to managers.
- K. Bowles: The question was raised earlier, "How can the universities be involved?" One thing we're going into that may be relevant is a collaborative operation with UC extension in San Diego which has an office of national media courses. What they're currently promoting and distributing on a mass basis is courses packaged using television through public broadcasting. The other medium they are using is through newspapers in which they have many books which are distributed. In both cases the courses are supported with package materials for instructors or administrators largely catering to the needs of the community colleges. McGraw-Hill has a national sales force that goes out to the book stores in the colleges and knocks on people's doors to promote this package. Extension has a contribution to make in that case, as it provides an aura of legitimacy to the courses.

[Bowles goes on to describe the marketing activities for his computer based Pascal-instruction system.]

The software that we're using is similar to the software you will be building in your language. Pascal is a similar language. I would think that this approach could be applied more and more specifically to the propagation of DOD1. As the details of the language become available it wouldn't take very long to package materials to be promulgated through outlets of this type.

- T. Standish: I happen to be a close friend with the computer

science editor of one of the major publishers of computer science and they're doing some very hard crystal ball gazing about whether or not they should get into the business of software production and maintenance, along with their text books, manuals, and other such things. The question of whether they should get into computerized, interactive media of learning is related to the possibility that books may become obsolete a few years down the pike under competitive pressure from TV terminals that contain an entire hour of color programming. They suspect that the competition for books is going to get very severe, particularly with people who were brought up on six hours of TV a day, and they're looking with some wisdom and foresight about starting divisions which will get into the software development and maintenance game for these educational devices. The picture commercially may be ultimately very rosy in finding a receptive audience of people willing to put up their own venture capital to go through a training medium type exercise for this language, whatever it shall be called.

- R. Kling: I just want to go back to some of the comments made by Santoni and Col. Whitaker. It seems to me you're saying there are often no incentives for managers to train their staff in use of the appropriate language practices. The biggest problem I see with DOD1 is that many people expect it to be used seriously in the Fort Dix's of the world, not the University of California's of the world. One of my RA's is doing a study of programming environments in one of the major insurance firms here in Orange County. He was looking at systems in use and he studied the software tool environment. He found several books on software tools in their library -- unopened. This is a firm with a fancy office over Newport Center, a good national reputation. It is a place that is known as a state of the art, "leading edge" software environment within the insurance industry. And that is a great place, not a mediocre place. I think it's really important to understand what the Fort Dix's are like, or the equivalent. It is also essential to understand what kind of training and incentives would encourage people to take DOD1 seriously.

Richard N. Taylor
Boeing Computer Services
Systems & Software Engineering Laboratory
P.O. Box 24346
Seattle, Washington 98124

Position Paper

Boeing Computer Services believes that the Preliminary DoD Common Language Environment Requirements document contains many important ideas addressing the systematic development of common language software. Several key issues, however, require elucidation and expansion. Chief among these are the concepts of lifecycle documentation, verification, and maintenance.

Under the heading of "Other Supporting Software" and "Project Management Aids," the preliminary document refers to tools aiding in requirements analysis, design, documentation, coding, verification, and testing (among others). A brief functional description of each is provided. Allusion is made to the widespread use of a development data base and complementary uses of the various tools. The impression left, though, is that such integration and sharing of information is stilted and shallow. In contrast, what must be reflected in the requirements is that the use of a system data base (containing everything related to a particular project), the pervasiveness of automation and verification, and the integration and pooling of capabilities must characterize the entire program development process, from requirements definition through code production. This high level overview provides the basis on which requirements for the individual tools may be structured.

Such a requirement will promote manageability through visibility, efficiency through automation, and maintainability through completeness of historical information. "Maintenance" should not be characterized as another step in the development process, since it encompasses activities identical to those in the development cycle. Further, testing must be regarded as a pervasive activity, occurring during all phases.

The system development data base will be composed of information produced by each component of the software development environment. Requiring formal, rigorous, machine readable output from each component will promote automatic analysis and report generation. Such information provides penetrating visibility into each phase of development process.

Requirements definition, preliminary design, detailed design, and coding are all subject to two types of analysis, both of which must be performed at each step to ensure correct progress. The first is for internal consistency and the second is congruence to the previous phase in the development cycle; requirements are verified back to the end user [Figure 1]. Automation may be employed for the majority of this analysis. Delay of such analysis only delays detection of errors.

An important feature to note is that the structure of the analysis should be essentially the same within each phase, only the detail and external representation of the source text change. In a programming environment separate front ends may therefore be provided for the requirements, design, and coding representations, producing a single representation upon which analysis routines may act. Note that the relative efficiency of the techniques employed may vary with the design detail.

Happily, the several techniques comprising the analysis facility promote integration. Static analysis, symbolic execution, formal verification, and dynamic testing all possess mutually complementary strengths and weaknesses. Exploitation of such potential must not be overlooked in the requirements. An overview of what such a system might look like is given in Figure 2.

In summary, careful consideration of lifecycle issues will provide correct guidance in the establishment of an effective program development environment. Further consideration of the ideas in this paper may be found in the reference.

Reference:

- L. J. Osterweil, "ASSET: A Lifecycle Verification and Visibility System," Proceedings of the Navy Conference on Software Specification and Testing Technology, Falls Church, Virginia, April, 1978.

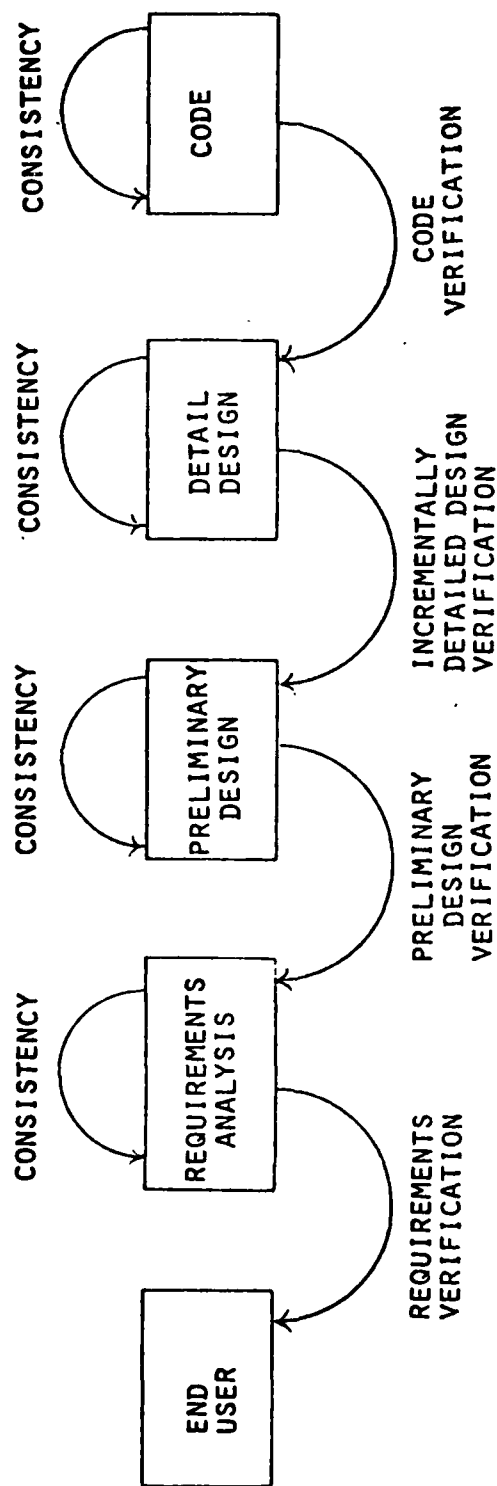


FIGURE 1: LIFECYCLE VERIFICATION

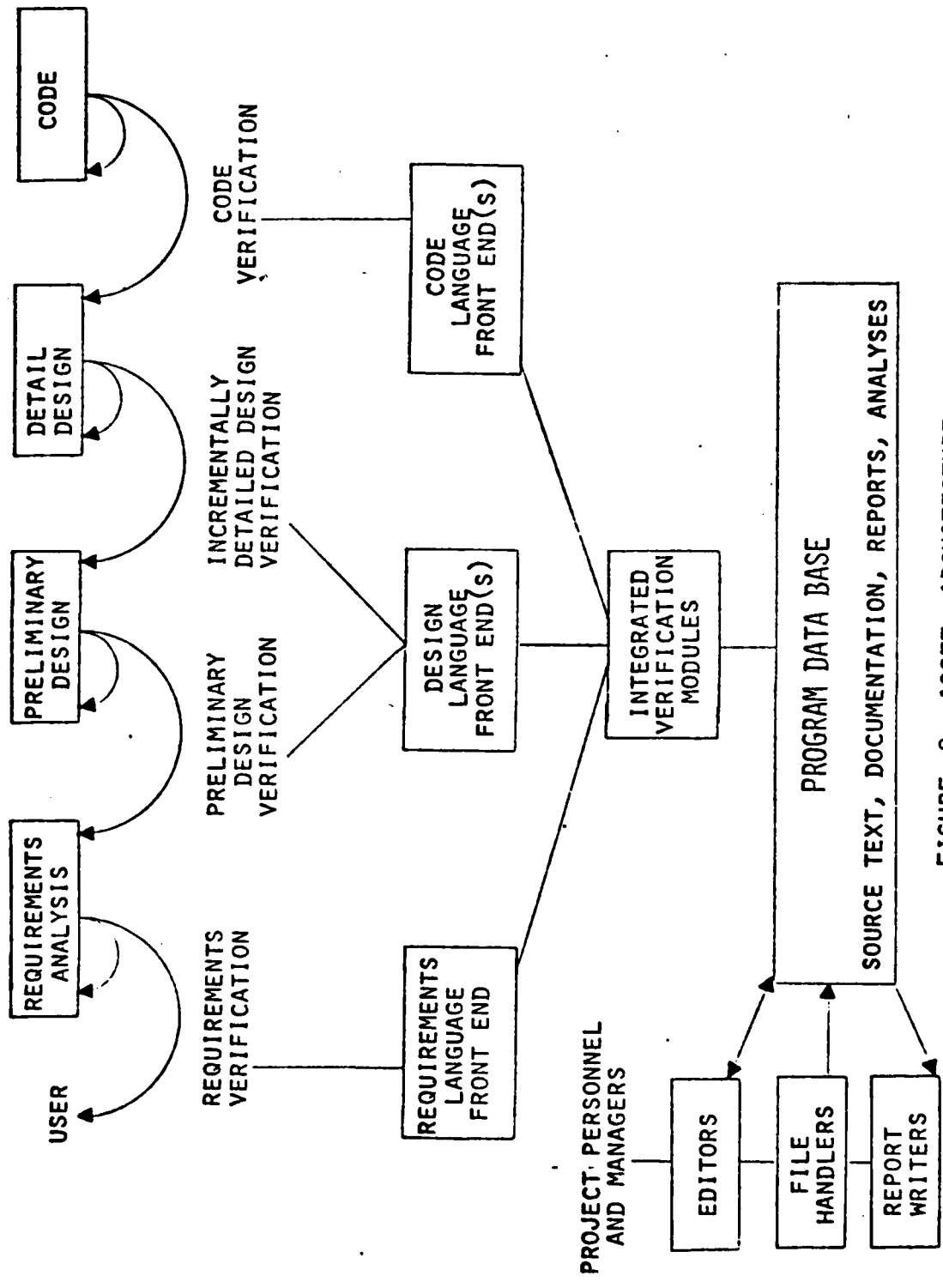


FIGURE 2: ASSET ARCHITECTURE

DOD COMMON HIGHER ORDER LANGUAGE
ENVIRONMENT WORKSHOP
POSITION PAPER

from: John Burgey
John Machado
John Perry
Patricia Santoni

The objective of the following is to put forward the concepts on which we believe an embedded computer systems support environment should be based. They fall basically into two categories. First is the development of a set of integrated tools to support the requirements specification, design, development, test, maintenance and management of DoD embedded computer systems using the common HOL. Second is the creation of a center for the distribution and configuration management of the resulting DoD HOL support environment.

The following paragraphs illustrate the sorts of tools we believe to be pertinent to such a support environment. This is followed by the fundamentals for the establishment of an effective management and distribution plan.

Complete, consistent software requirements specification for embedded computer systems are a necessity. They are facilitated by the use of requirements specification and analysis tools such as:

- Requirements Specification Languages - unambiguous, non-procedural languages which allow the system analyst to rigorously express the requirements of the system under development. Communication between participants in this phase of the development is made much easier and clearer by the use of such languages. They also provide the basis for tracing requirements throughout the development cycle and for validating and verifying the system against its requirements. Examples of such languages are the User Requirements Language (URL), the Problem Statement Language (PSL), and the Requirements Statement Language (RSL).
- Requirements Language Processors - computer programs whose input is text written in a requirements specification language. These programs analyze this text for syntactic correctness and create a data base which records all that has been stated concerning the system under development. All processors provide the means for analyzing this data base and issuing reports which provide information on the completeness and consistency of what has been said about the system under development. Some also provide requirements simulation capabilities, allowing the analyst to determine whether the implementation of the requirements is feasible. Examples of such processors are the User Requirements Analyzer (URA) which processes URL, the Problem Statement Analyzer (PSA) which processes PSL, and the Requirements Engineering and Validation System (REVS) which processes RSL.
- Requirements Simulation and Modeling Tools - computer programs which provide the analyst with the ability to model his requirements in order to experiment with trade-offs and to evaluate the viability of the requirements set.

Careful design of computer systems is one of the major thrusts of software engineering in the DoD. It is supported by such tools as:

- Design Specification Languages and Analyzers - languages which are used to describe the system designer's design for a system. They are rigorous, computer-processable languages which may be procedural or non-procedural. Their accompanying analyzers check for syntactic correctness and

output a variety of data bases, reports, and drawings. Examples of such tools are SPECIAL, AXES, and PDL.

- Design Methodology Tools - computer-based tools which support and enforce the concepts of the associated design methodology. Examples of such tools are the Hierarchy Manager, a Module Checker, and an Interface Checker which support HDM.
- Design Modeling Tools - computer-based tools which allow the designer to evaluate his design for performance and to perform trade-offs. Examples of such tools are those proposed by the Performance Oriented Design (POD) project and various discrete and continuous modeling tools in existence already.

The majority of tools available today fall into the implementation category. Some are planned for development for common HCL already; others will have to be provided before any work in this language can be performed.

- Compilers and Code Generators - the initial compiler available in this support environment will be the common HCL compiler developed by the HCLWG. It will also provide a number of code generators as they are developed, all of which will be distributed through this facility.
- Loaders - programs which resolve external references and assign absolute addresses to relocatable code, so that the resulting code may be loaded into memory. Loaders accept as input the output of one or more compilations and produce, as output, an operational program.
- Linkage Editors - programs that are used to determine which modules of a program system need to be loaded together. They also provide various manipulative capabilities for preparation of the load image.
- Flowchart Generators - programs that accept, as input, a source program and produce, as output, a flowchart of that source program. They help one determine the structure of existing programs and provide compliance with various requirements for program documentation.

Rigorous testing of all software elements prior to checkout on the actual target hardware is the motivation behind the development of an advanced set of testing tools in a support environment.

- Target Machine Emulators - programs that emulate the operation of the standard military computers to allow for unit testing of software modules on a host machine. Testing with the emulators makes it possible for many programmers to proceed with testing simultaneously without requiring exclusive access to the target hardware. This allows the programmers to eliminate many software errors early in the development cycle. They also make it possible to employ sophisticated debugging techniques which are, often impossible on the real machine.
- Program Debuggers - programs or routines that provide

THIS PAGE IS LOANED FROM THE HCLWG TO YOU
FROM OUR FORTHCOMING TO YOU

diagnostic information useful in locating errors in a source program. Debuggers commonly provide the following kinds of information: dumps, which record the complete state of execution (memory and register contents) at some point, usually the point of termination; snaps, which record intermediate values of certain items during execution; traces, which record the state transitions that occur during execution; and breakpoints, which interrupt normal computation and cause debugging activities to commence. Debuggers may operate either in batch mode or interactively.

- **Dynamic Analyzers** - programs that augment source code by adding counters and various other statistics-gathering indicators, then execute the augmented code and produce reports on how thoroughly the various portions of the source code have been exercised. These tools aid in determining when source programs have been tested adequately, and what kind of test cases need to be submitted in subsequent runs. In addition, this information can be used for determining frequently-exercised program modules, so as to isolate the most valuable sections to optimize. Many dynamic analyzers have been developed; they are frequently referred to as program path execution analyzers.
- **Static Analyzers** - programs that compute statistics based on the number of times various items appear in a source program. These are growing in use and are frequently used to derive a measure of "program complexity". Static analyzers are often referred to as program statistics gatherers.
- **Test Case Generators** - programs that help the user discover what input data will exercise a specified path in the program. These tools are still experimental, in that they frequently fail to generate test data that verifies that all program paths have been tested.
- **Interface Checkers** - programs that examine data utilization across separately-compiled modules and determine whether the accessing module interprets the accessed information in the intended fashion. As the goals of program modularity and reliability are more widely achieved, interface checkers will see greater use.

All of the tools that support the analysis, design, implementation, test, and management phases of software production are generally applicable to the maintenance phase as well. However, in the software maintenance phase there is frequently increased emphasis in the areas of configuration management and control of new software releases, program trouble reporting, software change requests, training, and documentation. Tools that are particularly associated with the maintenance phase include those for:

- disseminating information to users
- change request and trouble reporting
- automatic documentation updating
- user training aids and manuals

THIS IS A BEST QUALITY REPRODUCTION
FROM GPO COLLECTION TO HQ

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM GPO FORM 10-100

- . distribution of software releases
- . configuration management record keeping.

The objective of the management tools is to provide project managers with the means to plan, monitor, and control the work of their programming staffs by achieving a high degree of management visibility throughout all phases of software production.

- . Management Report Generators - programs that collect data and generate reports on the status of the software, the resources that had to be expended, and the activity necessary to achieve that status. This data provides a manager with the means to define and control the software tasks, assess incremental progress, detect trends and anomalies, and assure observance of instituted policies and procedures. The reporting system can be structured to obtain summaries by meaningful categories that can be compared with initial schedule, manpower, and cost estimates; this management data can additionally provide a statistical base for studying programmer productivity issues.
- . Configuration Management Aids - programs that collect configuration identification information and historical data that can be used for baselining purposes, design change control, software release, auditing, and tracking changes throughout the software life cycle.

In addition to the above tools, it will be assured that, in order to perform the best possible work, the designers and developers of any common HOL system will have available to them the following sorts of tools: an on-line help facility for assistance with the use of tools, text editors, library processors, common utilities such as file maintenance and copy capabilities, and adequate training and documentation.

The degree of success of the common HOL program will depend on the approach taken for the management of the maintenance and distribution of the HOL software development system (i.e., compiler, operating system, and tools). We recommend the following approach. The various compilers, operating systems and supporting items are developed and sent to the DoD HOL distribution center. The center will have an automated means (e.g., the ARFANET, AUTONIN II) for the distribution and computer-to-computer download of the items. Once an item is received at the distribution center, the center responsibility will first be to test the new item to assure that it will function when released to the user. The function of the center could be extended to include user-provided programs and a catalogue to aid other users in locating these. Next, the item will be made available to the general user, probably by an automated announcement. The user can then obtain a copy (in object code). There will be an automated vehicle for the request and transmission of the item. The center will maintain a list of each item requested by each user. Another responsibility of the distribution center will be to provide an automated error reporting system. That is, all the DoD HOL system malfunctions are sent to the DoD HOL distribution center. The center then contacts the responsible development group and broadcasts information about the malfunction to the appropriate users.

The distribution center functions as a focal point for all system software and related items. It is the only source for automatic training aids, documentation, information, compilers, operating systems, and tools. All error reporting, suggestions, etc. must be routed through the center.

John Purbey

John Machado

John Perry

Patricia Santoni

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO BDC

by Robert Balzer

Level 1: Non Interactive batch oriented debugging

Language-Support: None

Technique: Insert "debug-time" print statements in program

compiler switch

Technique: Source level trace

Run-Time-Support: Circular buffer for post-mortem dump

Technique: Insert breakpoints, check variable values, statement at a time execution (with on-line trace)

Run-Time-Support: User interface, symbol decoding

Technique: + check control stack

Run-Time-Support: + display of control stack

Technique: + change variable values and/or position within control stack

Run-Time-Support: restart computation, interpreter of assignment statements

Technique: Time-lever which programmer can control to run program forwards or backwards

Run-Time-Support: +

Technique: + immediate execution of extended language statements

Run-Time-Support: + interpreter

Technique: Callable external editor for modifying a program unit and continuing computation

Run-Time-Support: + ability to invoke editor and either compiler or interpreter on a program unit

Level 9: Structured program modification

Technique: Editing parse structure of language (ECL and Interlisp)

Language-Support: + useable version of parse structure accessible,
compiler/interpreter driven by parse structure

Run-Time-Support: Special structured editor

Level 10: Extensible debugging environment

Technique: User written tools which manipulate programs as data

Language-Support: + programs as a data type

Run-Time-Support: + accessible mechanisms (stack, error handler,
function definition, etc)

Level 11: Transportable debugging environment

Technique: Support system written in support language (DOD1)

Language-Support: + programs as a data type

Run-Time-Support: + accessible mechanisms

Additional Facilities

DWIM - Do what I mean (Teitelman)

Technique: Automatic error correction

Language-Support: + separation of identifiers into semantic classes

Run-Time-Support: + accessible error mechanisms

Programmers-Assistant (Teitelman)

Technique: Maintain session history; redo, modify, and/or undo previous commands

Language-Support: +

Run-Time-Support: + either undoable versions of all functions, or use of
extended trace facility

Advice (Interlisp)

Techniques: Quick source level patches to interface between modules

Language-Support: +

Run-Time-Support: Breakpoints and interpreter

File-Package (Interlisp)

Technique: Maintain updated source and object files as programs are changed

Language-Support: +

Run-Time-Support: Advise facility (to keep track of programs
modified by editor)

Robert M. Balzer

Maintenance

Not part of Pebbleman

Military Definition - all modification activity that happens
after delivery

Existing Characteristics -

dominant component of life cycle costs - 80%-90%
least desirable aspect of programming development cycle
normally performed by different groups than developers
difficulty increases as system ages

Belady & Leman - System Growth Dynamics

- maintenance reduces system structure
- maintenance cost is an exponential function of
lack of structure
- discovered cases of systems going critical

Causes of Maintenance Problem

1. Obvious technological answer;

No technology for maintenance if program development
is a cottage industry then maintenance is at the
stage of nomadic wondering. Maintainers must under-
stand how program is structured (design information)
and how a change will affect that program (dependency
information).

First is a documentation issue about the development
history of the system

Second is a sophisticated perturbation analysis
currently resting upon the simplest syntactic tools
such as cross reference of variable usage and
modification and subroutine calls.

2. More fundamental answer:

Is that such maintenance technology cannot be created. The current maintenance paradigm itself is intractable. Reason: Optimization and Maintenance are diametrically opposed processes.

Optimization spreads information - increase inter-connectiveness and thereby completely reducing structure and comprehension.

Maintenance requires locality of information.

Fundamental management problem: At the logical level at which manager understands a system virtually every change is trivial but at the (optimized) code level none are. Obvious solution is to operate like managers and modify (maintain) the logical specification. This necessitates a new technology for reliably and inexpensively reimplementing the modified system. These requirements can be easily fulfilled if the developmental history of the original system has been adequately recorded. The recorded history can be replayed step by step until a step is reached which is either invalid or inappropriate for the modified system. The maintainer can then substitute an alternative step (elaboration) which further modifies the system and the replay continued to the next invalid or inappropriate step or until the reimplementation is completed.

The advantages of such a maintenance paradigm are quite significant.

1. Maintenance becomes an extension of development and operates in the PDS environment with all of the developers tools
2. Development history becomes the guideline for maintenance
3. System structure is maintained rather than dissipated

The requirements for this paradigm are the step by step recording of the development process. This implies a single wide spectrum language used for both specification and implementation and methodology of step by step elaboration to gradually replace the specification components by implementation ones through repeated elaboration.

A number of prototype systems matching this paradigm already exist, usually under the rubric of source to source transformation systems. Several are represented here: Cheatham, Loveman, Standish, and Balzer. Of these, Cheatham's is probably the most advanced, and he is already performing maintenance in this way.

TOWARD SELF-DOCUMENTING PROGRAMS

Edward A. Taft
Xerox Palo Alto Research Center

June 21, 1978

Wednesday's session on program documentation brought out a number of approaches to the development and maintenance of external program documentation (that is, aspects of a program that cannot be expressed in the programming language itself), and a great deal of work has gone into developing manual or semi-automated documentation systems that, to a greater or lesser degree, are integrated with the overall program development process.

Many people will argue, however, that if a programming language does not provide the means for representing the essence of a program in the language, the language is somehow deficient in expressive capability. While the goal of designing a language that is ideal in this respect is unlikely to be met in the near future, I would like to point out that we have made considerable progress in this area during the past 15 years or so. However, taking advantage of the facilities available in modern high-order languages requires some revised approaches to programming.

Pascal and languages derived from it, including DOD-1, provide powerful primitives for program self-documentation through their facilities for strong typing and interface control. In particular, the distinction between declarations about the abstractions realized by a module and the procedures by which those abstractions are implemented is crucial. Ideally, the user of a module should be able to learn all he needs to know by reading only the declarations.

One powerful tool is type articulation, that is, the ability to manipulate objects with distinct abstract properties without regard to their underlying representations. The ability to provide distinct definitions of independent abstract objects that may or may not happen to share a common underlying representation is extremely useful. However, becoming proficient at describing abstractions through the type calculus requires some significant changes in a programmer's approach to implementing those abstractions.

The second important technique is that of hiding representation of and controlling access to objects through careful interface design. Since it is not now possible to completely describe the behavior of an abstract object as a part of the definition of the object itself, one may instead control the behavior of that object by defining a limited set of operations on the object. Knowledge of the representation of the object and the implementation of those operations is localized to the defining module itself.

While these are hardly new ideas, it has been only in the past few years that system programming languages with the necessary capabilities have started to see wide use. Therefore, it is not surprising that strategies for making effective use of them are not yet fully developed or widely known. They are not obvious but require much careful thought and practice. I suggest that you refer to [1] for some elaboration on this topic.

A bit of historical perspective is in order here. Early high-order languages (particularly Fortran) provided a flat control structure and very limited control primitives.

Therefore, external documentation of a program's control, in such form as flowcharts or English narrative, was quite important. Over the years, the control primitives of modern languages have developed to the point where many people feel that the language itself is a superior medium in which to document the control flow of a program.

The Pascal class of languages has the potential for enabling a similar advance in a different domain: describing the behavior of abstract objects. It is essential that we take advantage of this opportunity while considering the design of documentation standards and tools for the DOD-1 environment.

- [1] Geschke, Morris, and Satterthwaite, "Early Experience with Mesa", Comm. ACM, vol. 20 no. 8, August 1977.

22 JUNE 1978

Program Development Systems - An Overview

T. E. Cheatham, Bob Balzer, John Esch, Robert Morris
Ann Marmor-Squires, Stephen Squires, Tim Standish, Ed Taft

This paper presents an overview of a Program Development System (PDS) and a description of the tools required initially to do effective and efficient program development. Since we can also forecast that new tools might be available or might be required in the next few years, we want to make sure that new tools can be integrated as they become available.

To this end, we begin by talking a little about the programming process. We need to lay out just what sort of help we expect from the kind of system that we are proposing. A compiler does not exist in a vacuum. The notion that the programming process consists of an iterative process of source program preparation, compilation, execution, and debugging is not a dead notion, but it leads to a completely inadequate model of the appropriate tools for program development and the way in which these tools interact.

We start with specifications. Perhaps these are tight and precise, but more likely they are a loosely understood collection that needs continual refinement. We start a process of decision making and the decisions are often made incorrectly and have to be remade later, decisions to choose certain kinds of representations for data, decisions to choose certain kinds of implementations for operations or transactions or algorithms. We might try to visualize the whole process pictorially:

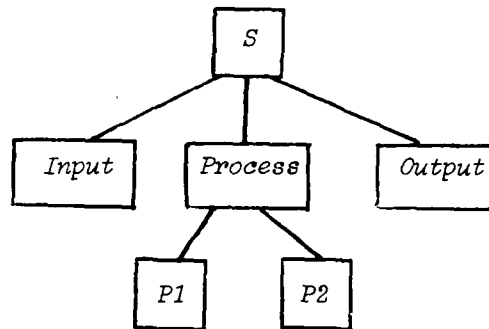


Figure 1

Basically we start with a thing *S* to solve. We chop it into pieces, perhaps the input, the process itself, and the output. If we decide Process is too hard, we might divide it into process one (*P1*) and process two (*P2*). We go through a procedure that in the end gets us some things --- presumably modules in DOD1 which are subject to compilation, loading, and execution on some computer.

So we want to talk, in some sense, about a system which lets us represent or contain or implement the boxes and connections in Fig. 1. A system that lets us get into a context of dealing with these boxes at a variety of different levels. The system should let us move among the levels of a program from the very abstract levels that says "solve problem" all the way down to the load modules. And surely a Program Development System should be something that lets us retain and manipulate all these levels of program.

Now let us state a few assumptions we think we should make. The first is that we suppose the programmer has access to an

AD-A089 090

CALIFORNIA UNIV IRVINE DEPT OF INFORMATION AND COMP--ETC F/G 9/2
PROCEEDINGS OF THE IRVINE WORKSHOP ON ALTERNATIVES FOR THE ENVI--ETC(U)
1978 T A STANDISH

DAAG29-78-M-0219

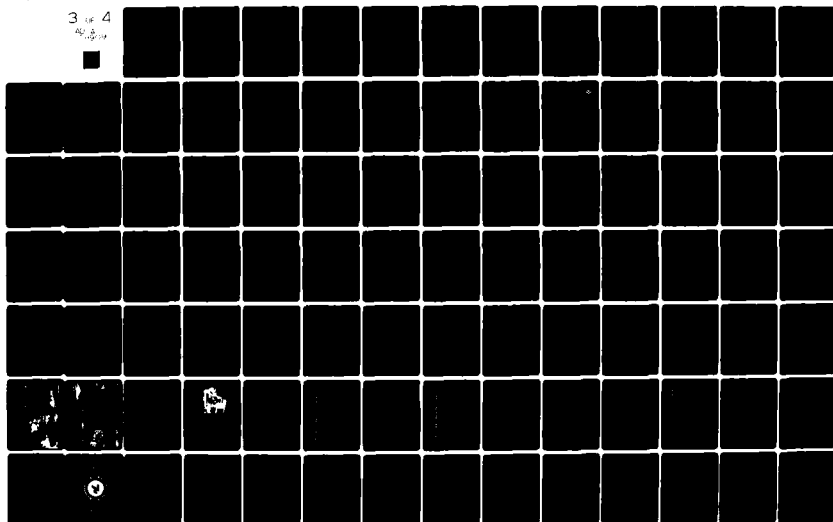
UNCLASSIFIED

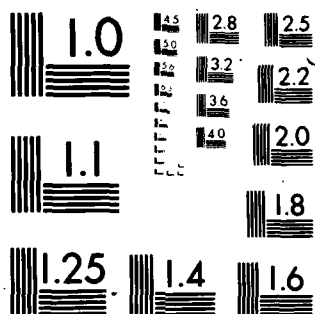
UCI-ICS-78-83

NL

3 of 4

AD-A089 090





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

interactive system of some sort. Second, we should assume that that system is of sufficient size to host a Program Development System. That obviously means it's not the computer in the nose cone of a missile. It means it is something at least the size of a small 360 or a PDP-11 or whatever.

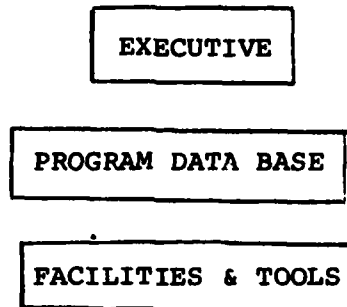
One of the first questions, of course is: "How do all my programmers get on that there thing? --- I haven't got one." As an aside, we should like to make a couple of remarks about the National Software Works. Such an environment of interactive tools can exist on a network, with a smooth interface to the user, and can host interactive program development systems. The NSW can also provide access to these tools without having to make project managers spend money from their own development and project budgets to capitalize the Program Development System host system --- a distinct advantage. The National Software Works now exists in some sense --- in the sense that there is a Works Manager and there are actually tools available. At this very instant in time, it is viewed as a little klutzy in the sense that the communications go a little slow for some tastes. But the point is that the technology is in place right now to permit people to employ reasonable computers to do program development. This is the basis for accepting assumptions one and two --- that people can have a reasonable size machine.

Another thing we have to contend with is the masses of naive programmers. The NSW, in some sense, deals with deprived programmers, ones who do not have a sensible machine or sensible tools at their disposal. The NSW surely demonstrates that it is possible in principle for them to have one. But what about

the kid that came out of high school and took two weeks of training in the city and is trying to code something. We frankly don't know what to do about him. We think the kinds of tools we're talking about are a little too sophisticated, but we don't know. Programming is not a task that can be done efficiently by acres of more or less incompetent and naive people.

It has been the experience of each of us, and we could cite that experience, if challenged, that smaller groups of more competent programmers with proper tools can outproduce the thundering hordes by orders of magnitude. What we're talking about are systems that enable the more talented and more professional programmer to have access to tools and facilities that let him get about his real business. We can assure you that it is much easier to build a major system with three or four people than with thirty or forty (many of us have done both).

We shall now sketch briefly what our conception of a Program Development System consists of. We can look at it as having three components. Then the question becomes what of its pieces should be standardized, required, and carefully specified at this point in time or very soon. We would like to imagine that the initial experience with DOD1 will be in the context of some reasonable collection of tools and components of a PDS.



3 Components of a PDS

Fig. 2

One component is the *Executive*, the thing to which you talk when you sit down at a terminal and say, "Hi there, I'd like to do something." There is some entity to which you are saying "hi there", and that entity presumably knows about you and knows about what files you might want to deal with and it knows about the tools and facilities you might want to call up and knows about how to manage all this in some nice, sensible, coherent way.

The second component, a crucial one, is what we might call the *Program Data Base*. The view here is that you approach the computer to make a few transactions----"Hello, I'd like to edit my procedure SAM, I've got a few more things to say about it. Oh, by the way, I'd also like to run my other program HENRY." You do a series of transactions to update, in effect, the program data base --- correcting things, defining things, adding things, augmenting things. It all resides on the computer and in such a way that you can get at the parts, you can name

the pieces, you can work with them. We are certainly assuming the programs themselves are included in the program data base.

We have to say a few things about the programs. We suggest a hierarchy of levels of representation or levels of abstraction of a program with the bottom line being the program modules in DOD1. We've got to talk about the ways in which a PDS goes about house-keeping these modules; house-keeping the relations amongst them and letting us move around this turf somewhat. Programs will be written in DOD1 and, at the very least, also in a form of DOD1, in which, certain things are missing and are replaced by English descriptions of what they will later become.

We think it also important to imagine that the Program Development System Data base have some history --- sufficient history to let us know what happened, and when, so that we can fall back to some previous point, and find out how we got to where we are. We all agree that maintenance consumes the lion's share of the total system cost over its useful lifetime, yet we find that the normal course of things in embedded systems is that you are maintaining a box of binary cards, and that an act of maintenance means making up a few patches that hopefully don't screw up more things than they fix. Now, there are reasons for the high cost of maintenance. One of the problems there, in our opinion, is that when you are presented with an act of maintenance to perform, what you have to do is to redo something that was done originally in a different way --- so what you'd like to do now is to get back into the context in which the original decision was made (perhaps you allowed three bits for this field and now it requires four bits, or at a higher level it may be "this thing used to have this characteristic, and now it doesn't") but whatever decision was made, to accommodate for change, one has to remake that decision in a different context and follow the effects of that change down to the actual operational code.

Now in a PDS one would hope that one could arrive at the proper level of abstraction, go in and find out what the ramifications are of some change we are going to make, make the change at the appropriate level and hope that the

final load modules can be regenerated in some more or less mechanical way. So we have to have these various levels of representation of programs and we have to know how we can go from one to another. The process of getting this refinement has to be recorded in the data base, so that we can replay the generation of a new system, after we have made changes at any level.

The third component of a Program Development System is the set of *facilities and tools* that reside there. Let's just recite a few of those tools to suggest the kind of things that we're talking about. One of the things we want to reflect on are the kind of hooks and handles required, and the patterns of coordination in the overall system and in these various tools to form a coherent and cooperative PDS, in which these tools can work compatibly together.

One of the very important lessons to be drawn from our experience with environments is the beautiful way in which the tools in such systems can interact with one another. But that doesn't happen automatically; it requires a great deal of planning and foresight. One of the important questions to be answered at this time, then, is what can we say about tools and Program Development Systems that will let us, over the next decade or two, add more elaborate and sophisticated facilities. More fundamentally, one of the questions we should like to answer is twofold: (1) What are the basic elements of a Program Development System we should have, in effect, on day one, and (2) How do we prepare the way for gracefully and easily adding tools over the next decade or two.

Now let us reflect on some of these tools. One of the obvious tools is something to edit with. Editors can appear in many ways in this modern world. You can have editors that are very smart about what they are editing and that understand the structure of programs (and that's a much nicer environment than an editor that just knows about text). But an editor is used to do many things --- to change things, to define things in the first place, and we can talk about editors that prompt you to pull things out of you, versus editors that are passively sitting there waiting for you to spill characters at them, and so on. Editors are the obvious tool to produce and maintain documentation.

Another thing we can talk about as a tool is one which lets you refine from one module to its descendant -- in the sense of a refinement being a decision about chopping into pieces or a decision to choose some representation of data or program, or to say this concept that I was fuzzy about before is refined now as this more detailed thing. What are the facilities that let us do these acts of refinement?

Other kinds of tools we might refer to as analysis tools. There are a large variety of tools for analyzing. The most simple of these is very likely the one which says "is this program syntactically acceptable?" This is a very, very modest kind of analysis, and, indeed, one could argue that a sensible program editor in 1980 should have that ability built in. Another kind of analysis is what we call type checking. In a hard-typed language like DOD1, one of the semantic things you'd like to know about your program is

that the types are all compatible. If they aren't, there is not much point in compiling and trying to execute. The next kind of semantic analysis might be to examine certain classes of potential faults --- for instance, looking at a program and saying "well, everything is all right with regard to subscripting except right here, where it looks like you're going to have a problem".

That kind of analysis tool is doing a more sophisticated interpretation of the intention of the program. Thus, we can imagine a hierarchy of analysis tools, each looking deeper and deeper at the semantics of a program, to determine the presence of certain good features or the absence of certain bad ones.

Another kind of tool we might talk about is one to establish some environment so we can run a program. There may be packages that will provide you with behaviors that replace incompletely specified modules or software that simulates characteristics of external sensors and devices. Of course, we want to execute programs. There is nothing quite like trying to execute a program to see what happens when it runs.

We often want to probe a program. A probe can do many things --- we might want to probe to get some assessment of where a program is spending its time (this is a cost probe). We may want to probe it to talk to us (often called breakpointing or tracing) --- so the program says "Hi, I got here. What do you want to do now?". Thus, there are a number of kinds of probes which go all the way from collecting

data to informing the program to stop and report back to us, or whatever, that we might like to insert into a program as we make our first attempts at running and debugging it.

Surely, we want to make queries about the program --- "Who talks to this guy?" --- "Who does this procedure call?" In a large program, if we change "this", we'd like to know "who is affected."

Last, but not least, we might like to compile a program. The old model that you prepare a text file, you compile it, you load, and you go --- is just an inadequate structure for a PDS. For example, both compilers and analysis tools share certain parts of the structure --- what the flow structure is, who gets set where, what the types are --- if you follow this far enough, you come down to wanting to forget the notion of a compiler and have the notion that at some level you may want to take this program and generate code for some machine. This can in fact be well into the process of debugging and/or verifying your program --- you could have run it many many times before you ever think of compiling.

This is what we want to say to set the stage. Fig. 3 gives a table of the minimal set of tools we think the third component of a Program Development system should have.

Command Interpreter (Executive)
Editor
Parser
Some - Code Generators (normally more than one)
Program Executer
Probe Package (trace, break, measure)
Refinement Package (includes optimizer)
Program Queries -- (Who calls whom, etc.)
Test Data Generator -- (Simulates External
device behaviors)

Fig. 3

Last, but not least, we believe that the coordination amongst the tools must be extremely carefully arranged. For instance:

- (1) We want to be able to call the Executive from within any tool and get the Executive to call other tools in a nested fashion --- after which it cleans up and allows resumption from the point of nesting of the original tool activity.
- (2) We might want to arrange that tools can call each other (with the user playing the role of a subroutine which is called by the PDS and supplies values or commands in reply) so that tools can be cascaded in very powerful ways.

We know from our own experience that the technology to do the initial level of a Program Development System as specified in Fig. 3 is here now, and is of proven value in program development.

We think it essential that some way be found for such a Program Development System to be implemented in connection with the DODI effort. We seriously urge that our view be adopted.

Ideas on Compiler Validation-Testing
Susan L. Gerhart, ISI
June 13, 1978

1. To be credible, the compiler validation effort must conform to the highest possible standards of known testing techniques and use existing testing tools to the utmost. Testing seems to be most successful when performed in the spirit of adversary conflict, i.e. the testing procedures do everything known possible, within reasonable cost limits, to demonstrate that the compiler will fail in some way, e.g. by aborting, producing bad code, failing to compile legal, but perhaps bizarre, programs, or accepting illegal programs. This suggests that the series of test programs consist of several forms:

a. Normal, useful programs. Good candidates for this class include the external library routines, particularly the application oriented software packages. Such routines will have extensive associated test data which can be used to test the quality of the code generated. For new target machines, these programs would need compilation anyway, so the validation effort can kill two birds with one stone. Additional programs developed for tutorial purposes which demonstrate aspects of the language not covered in these application packages can be selected to fill out this class. In other words, the normal programs used for validation can be used for several purposes, thus reducing the expense of validation test development and enhancing the credibility of the validation exercise.

b. Bizarre, illegal programs. Many people have observed that there are no better tests for the robustness of a compiler than the first programs written by students in a new language. Missing brackets, illegal symbols, ill-formed statements, missing separators, etc. are all too common, no matter how much syntax instruction has preceded the submission of the first programs. This is especially true for programmers who are familiar with other languages and experience negative carry-over to a new language. It should be possible to acquire a good set of such programs after a few classes of programmers have been taught. A compiler which blows up too easily has been carelessly designed and should not pass the validation tests.

c. Almost legal programs. The spirit of the common language is that many restrictions are compile-time checkable. It should be possible to take the set of normal programs used for testing the compiler and introduce perturbations which do not disturb the syntax but do violate various restrictions. The compiler should reject such programs with reasonable error messages.

d. Perturbed, useful programs. As will be mentioned next, it will probably be necessary to include additional variations of programs to cover all constructs of the language in the compiler validation tests. It should be possible to take the normal, useful class of programs and perturb them in various ways to get a wider exercise of the compiler. For example, programs might be de-optimized in various ways, loops might be de-structured into conditional and gotos, expressions might be broken down into smaller units, parameters might be written in different orders, conditional statements might have their branches reversed, etc. If all these perturbations are easily seen to produce equivalent programs, then the same test data which thoroughly exercised the test programs (hence, the compiled code) should provide a reasonable exercise of the perturbed programs; of course the results should be equivalent. Such perturbations should also provide some measure of sensitivity of the optimization strategy of the compiler to various coding styles.

The wording of the Environment Requirements should be modified to include validation for robustness and ability to reject programs as well as "compile and execute a standard series of programs...". 2. The compiler should be instrumented sufficiently to demonstrate what is now accepted as the minimum acceptable coverage, namely, every program predicate exercised at least once, which implies every statement exercised at least once. It would be unthinkable to consider a compiler validated if this coverage criterion had not been met. Indeed, this requirement should be applied even before a compiler is submitted for validation with the standard tests.

The standard tests may in fact fail to satisfy the requirement necessitating additional test programs or modification to the compiler. A typical situation where this difficulty might arise is an optimization which just never gets invoked in the standard test programs. The optimization may be of questionable use and therefore removeable or it may just not yet have been required. Such a situation would require negotiation toward final certification of the compiler. Any compiler submitted for validation which contains logically unreachable code should be rejected immediately as carelessly designed and tested.

3. The test strategy suggested by 1 and 2 is based on the following observations about testing in general:

- a. Testing only for coverage is too weak a measure to ensure much reliability. This claim is theoretically and experimentally demonstrable.
- b. Testing from specifications alone (legal and illegal programs) is likely to leave some parts of the program uncovered.
- c. The most effective way of selecting test data is first from specifications followed by additional selection for uncovered parts, as shown by instrumentation results on the specification-selected data.

4. The full range of testing tools should be brought to bear on the validation effort. There may be tools specifically useful for compiler testing, either existent or easily developed. These should be explored and evaluated for merit in the context of validation.

5. A few experts in the theory and practice of testing should be brought in as consultants to evaluate the final selection of test programs. Again, the philosophy is to make the tests tough, while not letting the expense get out of hand. There is growing expertise in testing which could help meet these goals.

6. However, the set of test programs might be flexible. For example, experience (or insight) might show that some test programs are redundant and therefore can be safely removed from the test set. Or, especially devious programs might be developed by individual compiler implementors or discovered in teaching or use. Any programs which reveal compiler errors after validation are especially good candidates for inclusion in further tests. In other words, the initial validation test sets may represent minimum standards which can be increased with experience. Indeed, compilers might be required to be re-validated periodically.

Ideas on Compiler Validation - Proving
 Susan L. Gerhart, ISI
 June 19, 1978

1. There are several possible properties to verify about a compiler. One such property is that the compiler "accept" a program if and only if it is legal. Of these two aspects, it is probably more important to verify that if the compiler accepts a program, then the program is legal. In the other case, users can be expected to complain. Syntactic legality (purely related to parsability) is probably not worth formal verification, since parsing is so well understood, almost automatable. Other aspects of legality, e.g. aliasing, scope, type consistency, etc. are less well understood for both programmers and compiler writers. Indeed, there may be some very subtle semantic misunderstandings here, which makes this aspect of legality worth formal verification.

The second property is, of course, that the code produced for an accepted program is semantically equivalent to that of the source, assuming correctness of the run-time package.

The third property is that the run-time package is correct.

A fourth property is that code will be produced for a legal program, provided that resource constraints are not exceeded. Again, failure to produce code will result in user complaints, so this property may not warrant verification, at least not initially.

The most important concern is clearly that the execution of the compiled code match the intent of the source, which of course is the subject of further verification wrt its specifications. The point is that there are many properties which a compiler should possess, but they don't all require formal verification, or perhaps even verification at all. The task of compiler proving can be factored by these properties.

2. The task of compiler proving can be factored further by the identifiable modules of the compiler. There are data structure modules, e.g. expression trees, run-time and compile-time stacks, tables, code templates, etc. and algorithms, e.g. parsing, register allocation, run-time and compile-time storage management, code optimization, type checking, etc. Again, not all of these may require formal verification; testing, code walkthroughs, adaptation of published algorithms, etc. individually or together might suffice.

All of these components have been extensively studied for traditional compilers, although some new compilation techniques might be needed. The barrier for compiler proof is the same as for other large, well-understood systems, namely, that the knowledge about these systems has never been organized and stated sufficiently formally that proofs can start right off. Instead, there must be a lengthy period of identification of and development of notation for fundamental concepts, organization of operations, splitting up representation from abstraction, separation of optimization from fundamental operations, etc. All leading to the development of formal specifications and bases for proofs. Proofs can proceed along with these developments, even drive them, but a satisfactory proof will only be achieved after the underlying programming knowledge has been elicited, organized, and formalized. Historically speaking, this is the natural final phase of development of

scientific knowledge, freezing via formalization. In computer science, this is unlikely to occur unless forced so by formal specification and verification; it requires a lot of work and seems to be a long sidetrack from the immediate goal. But after the initial effort is invested in formalizing the concepts, the next efforts need only modify or extend the previous formalizations.

3. Consider the task of proving that the code generated from compiler C, written in language Lc on machine Tc, for source language S in target language T for program P is correct. That means that there must be semantics for at least both S and T. As pointed out by Hanan Samet, the task can be accomplished without Lc or Tc semantics if it is performed anew for each P. That is, the task is one of proving equivalence of P as written in S and translated in T. Equivalence is usually a reasonable proof exercise, although modern verification systems are not equipped to handle it directly. Indeed, this is the most effective strategy for microprogram validation. But the more efficient verification process is, of course, verifying the compiler for all programs. Tc might not be any real machine, but instead an abstraction concocted to aid portability. This would enhance the effectiveness of verification, but, of course, would introduce the extra step of verifying the end phase of specializing to individual machines.

More specifically, there are various strategies for proving some high level aspects of compilation, for example:

a. Prove that for every statement S, $P\{S\}Q$ implies $(\text{Compile}(P)\{\text{Compile}(S)\}\text{Compile}(Q))$; that is, the verification conditions at the source level imply the verification conditions at the target level. If the compiler has no inter-statement interaction, then knowing what the compiler would generate suffices.

b. Another approach is $\text{Target-interpret}(\text{Compile } G, \text{Data}) = \text{Source-interpret}(G, \text{Data})$, for program G and arbitrary Data. While this works as well as the first for simple compilers, it is not clear what happens when code generation is not straightforward.

c. Compilation can be viewed as a multi-stage transformation process from a representation using source-level constructs to one using target-level constructs. Another approach is to prove that each transformation in a compiler, whether between types of representation or within one type of representation, produces an "equivalent" program. The theory and technology of correctness proving is sufficient to support such a transformational approach to proving the correctness of a compiler, provided the transformations it induces are well articulated.

4. Is it feasible to prove the correctness of a compiler? Given the above ways of factoring the task, properties and modules, it seems so. Of course, this assumes that semantics of source, intermediate, and target languages will be available and described for verification purposes. Furthermore, it assumes that the compiler will be constructed using modern programming methodology in conjunction with the verification technology which supports it. The main barrier is not just size of the task, which can be factored, but formalization of the supporting theory of data structures and compilation algorithms to the point where verification is meaningful. This gap between understood, but unformalized, and the ultimate formalization should not be underestimated in its difficulty, nor in its practical value once achieved for further verification and construction of compilers.

Position Paper on Programming Environment

by Robert Balzer
USC/ISI

My main interest is in providing software tools to aid programmers develop and maintain software. In the short term much can be gained from exploiting the type of environment created in INTERLISP by Teitelman, et al. Although this environment has been specifically designed for LISP, almost all of it is transferable, given a careful system design. In particular a programming environment should include fully compatible compiler and interpreters, a program editor which operates on the structure (not text) of a program, Dwin facilities; an extensive interactive debugging facility; and a "Programmer's Assistant."

Over the longer term, we should recognize that programming using current approaches is a complex *manual* process performed in people's heads. This process has been completed, but not debugged, before they ever use computer tools. Increased reliability and productivity can never be achieved while we rely on unanalyzable manual techniques for such a complex task. Instead, we must develop methods for gradually evolving a program step by step from a formal specification as complexity, in the form of optimization, is slowly incorporated. Each step must be automatically recorded as part of the program documentation and verified to ensure the validity of the implementation. Only when programs are developed with the aid of such documentation and validation tools via a series of small understandable steps will significant progress be made in program reliability and programmer productivity.

The following abstract is from an ongoing research effort addressing these issues:

The Development of abstract operational programs is a major software research thrust. This proposal investigates how such abstract programs can be effectively and validly converted into efficient concrete programs. It proposes an approach called Transformational Implementation (TI) in which equivalence-preserving transformations are successively applied to the abstract program to effect the optimization. The key to this approach is that while optimizing transformations are selected by the programmer, the program is transformed by the computer system. Hence, the programmer designs the optimization; the machine ensures its validity and transforms the program.

In addition to guaranteeing the validity of the implementation this approach would also drastically reduce the time and effort required to implement a system. This would allow programmers to experiment with alternative implementations to widen their

experience base and improve their capacity to design optimization. Furthermore, maintenance would be performed at the conceptual level on the abstract program which would then be reoptimized rather than attempting to modify a complex interconnected optimized program as is current practice.

The purpose of this study is to discover: what facilities are necessary for the TI approach; what transformations are required; how are they expressed; what categories do they fall into; how can they be named; what kinds of applicability criteria are required; what processors are required to verify that these criteria are satisfied; what instrumentation facilities are required to test a transformation's selection criteria; what proof techniques are required to validate the transformations; etc. In short, the study will provide an experience base for the TI approach from which we can design and implement a prototype system.

Informal comments for the
Workshop on Environment & Control of
DoD Common High Order Language
June 20-22, 1978

Stephen D. Crocker
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

These comments are keyed to the statement of purpose of the workshop and represent my immediate thoughts upon reading the statement. For the most part, they represent accumulated prejudices I carry around on the general subject of compiler/language support.

Technology for Validation and Control

The most important step that can be taken in this area is to insist on a rigorous and formal semantic definition of the language. We all know, of course, that the techniques for formal semantic specification of a language are not well developed, and that the few serious attempts have encountered various troubles, but I'm convinced that the attempt must be made anyway:

1. A formal semantic definition is a prerequisite for formal verification of the implementation of a compiler. It is only a matter of time before formal verification techniques will be available. Verification requires a specification and the specification should be developed with the language, not after.
2. Even if the formal semantic definition is hard to read, a sufficient number of people will read it and understand it. In particular, compiler writers will understand it and base their implementations upon such an understanding.
3. It is quite likely that we will learn how to write readable -- even pleasing -- formal semantic definitions. Certainly if readable definitions are available, they will be the reference of choice for both users and compiler writers alike.
4. Some reference document is required. The reference document must serve as the basis for arbitration of differences in understanding that will surely arise. Even if an English or other informal document were more readable, the formal document provides a better chance for eliminating ambiguities.

These points address only the relationship between a formal definition and validation and control. It must be obvious that beyond these benefits, the definition of the language itself must benefit enormously if a formal definition is completed. I understand that formal definitions were suggested for phase 1 but not included for lack of time. If they are not forthcoming in phase 2, I suggest that they become a priority item immediately thereafter.

I view the formal definition as a trigger. Once it is available, a number of other steps may be taken:

1. **Verification efforts may be supported.** This is a research topic that is already partly explored, so the most relevant statement to make at this workshop is that the topic merits continued support.
2. **There are many ways to design a compiler.** If the formal definition is clearly in hand, then it is reasonable to examine the design of a compiler to see how the design relates to the language definition. Even without formal verification systems, it is quite reasonable to ask for documentation that is "verification-oriented" that relates the structure of the compiler to the definition of the language. Such documentation should be accessible to a wide audience. A large percentage of the bugs found in a compiler will probably be found by reading the detailed documentation. (The same remark about the quality of the implementation as the quality of the definition applies here. The design is likely to be much better if such documentation is required.)
3. **Short of verification, it should be possible to build a "reference compiler" based solely on the formal semantic definition.** The reference compiler should compute the correct results, but not contain any optimization for speed or space. When there is question as to how a specific program (fragment) is to behave, execution of it as compiled by the reference compiler may be taken as definitive.

In addition to an insistence upon a formal semantics and the several uses it enables, I recommend that a control office be established for validating compilers and that the major compiler vendors be queried for their experience in this area. With respect to the control office, the Cobol compiler testing operation in the Navy can serve as a standard model. Its approach is quite ad hoc, of course, but it serves as a mechanism for combining the collective wisdom of the user and implementation community.

Software Tools

Hooray! It is not yet commonplace for compiler writers to consider whether any tools should be supplied with the compiler. We must hope that the intentions of the sponsors are serious and that useful tools will indeed be built to complement the compilers. I do not have detailed data available, but I would not be surprised to discover that the lack of use of some of the prior DoD languages has stemmed from the poor quality or unavailability of adequate compilers and support tools.

There are two fairly standard kinds of support facilities that need to be supplied. Many others can be thought of, but I want to focus on two in particular.

First, the projects of greatest concern are the large projects. I don't need a quantitative measure to describe "large"; large projects are simply the ones that stress our technical and organizational capabilities. Large projects get in the most trouble and, of the projects that get in trouble, the large ones cause the most damage. While technical issues are often at the root of the troubles, I believe that organizational issues

are far more prevalent. Any kind of support we can provide for the organizational aspects of large projects is going to pay off handsomely. Among the obvious things we can do is provide flexible catalog and version control systems, document preparation and control systems for documentation, cross-referencing of documents with source and object code, etc. None of these ideas is new, but none is in use everywhere. One of the primary reasons these tools are not widely used is that most groups depend upon self-hosted compilers and support systems. This is a large mistake, I believe. The apparent saving in using a "free" machine is costly in productivity, ambition of effort and reliability.

Several systems built in the last few years attempt to provide support for groups of users building large systems. Many of these are proprietary systems used wholly within large companies. Again I suggest that input from vendors will be useful. Outside of the unpublicized systems, two systems have been documented in the literature. The Programmer's Workbench has been produced at Bell Labs and the National Software Works is under development with DoD funding. The NSW is particularly interesting because it combines the idea of supporting a set of programmers with a rich set of tools with the idea of using a national network for access to these tools. As I mentioned before, the organizational aspects of large project frequently dominate. These effects are magnified when the group is distributed across the country or around the world, as is common for DoD. The primary set of programmers may be grouped at one site, but the sponsor is somewhere else, the user in another place, the monitoring agency in yet another, and so on. Lack of close technical communication among these players has resulted in near total scrapping of some systems after they were delivered.

Even without the problem of coordinating dispersed groups, it is not uncommon to find that compromises are made in the design or development of a system for lack of access to existing but unavailable tools. For the kind of systems to be built with DOD-I -- computer systems embedded within larger systems -- the pressure to use the smallest configuration possible in the final product will almost guarantee that the self-hosted facilities will be less than the best. I believe that all aspects of design, implementation, documentation and testing should be carried out on equipment that is chosen for that purpose, and that the use of the product hardware be restricted to final testing. Sure, the cost will be higher -- on the surface. In fact, the actual costs will drop dramatically.

The NSW should provide uniform access to the best tools. Whether it will work out as well as intended is yet to be determined. But however the NSW fares, the basic lesson is that a separate and substantial support facility populated with the best available tools will materially improve the productivity and reliability of DoD systems. Without such support, promulgation of a new language is useless.

I have addressed the issue of access to tools. I now want to talk about a particular class of tools to support the individual programmer. We have known for a long time how to support a programmer while he tests and debugs his program, but we seldom supply the right tools. What are the "right" tools? The right tools are (at least) the ones found in Interlisp. Interactive execution. Tracing and breakpoint facilities without modification of the source code. Execution of mixed levels of compilation, including interpreted source code, individually compiled modules and block compiled modules. Editing of programs within the execution environment, thus eliminating

patching or expensive restart after editing. Extensive error checking and controlled error correction. Flexible interactive controls for running and rerunning program segments or for undoing the effects of a program. Measurements of time and space usage. Automatic formatting of program text to help detect syntactic errors and to eliminate concern with such details. [Also provides a standard format for all to use; promotes readability of code by others.]

Media of Communication

Another great question! The promulgation of a new language is always an uphill battle. Certainly one of the most serious problems is how to train people in the language. In the short run, almost everything will be required: short courses, textbooks, seminars at conferences, classes at DODCI and similar facilities, etc. In the long run, I think it will be possible to get the language taught in schools as a matter of course. The key will be in the government's attitude. In my view, it is entirely appropriate for the government to sponsor the widest possible use of DOD-1. While the language is designed and justified on the basis of embedded computer systems, it is definitely reasonable to use it for a wide variety of other applications. To the extent that students learn and use the language in the course of their studies at school, the government will benefit directly in the quality and quantity of programmers available.

Textbooks aimed at the general student population should be written. Use of DOD-1 in colleges and universities should be encouraged. Here the key will be availability of compilers for machines on campus. The usual large machines -- 360s, 370s, 6600s, 1108s, etc. -- and the usual small machines -- 11s, Novas, etc. -- should all have compilers for DOD-1. The microcomputer market should also be addressed. DOD-1 systems that run on 8080 microcomputers or their equivalent will be used enthusiastically and find rapid use by students and hobbyists. Active support or participation in the development of compilers for these machines should be encouraged.

Lest these suggestions sound too activist, I would never suggest that that government take any steps to actively discourage the use of other languages within the academic or commercial communities. Any such step would be met with the strongest resistance and eliminate any chance of substantial support for the language outside the DoD community. As a related footnote, widespread use of the language is more likely if the name is not "DOD-1".

file: dod-remarks

**DEPARTMENT OF THE AIR FORCE
AIR FORCE ARMAMENT LABORATORY (AFSC)
EGLM AIR FORCE BASE, FLORIDA 32542**



**STANDARD RETARGETING COMPILER
FOR TACTICAL WEAPONS**

1. Term Definition:

COMPILER: A computer program which produces a machine language object program from a High Order Language (HOL) source program.

CROSS-COMPILER: A compiler which resides in a HOST computer and outputs executable machine code for a different (TARGET) computer.

MODULAR COMPILER: A software implementation of a High Order Language (HOL) which is separated into basic modules. This paper considers the front end and the back end (code generator) as the only modules.

RETARGETING COMPILER: A MODULAR CROSS-COMPILER which has one or more optional code generators which allow the compiler to reside in one HOST computer but produce executable machine code for many other target computers.

2. Background:

a. Digital systems are rapidly replacing analog equipment systems in modern tactical weapons. New digital hardware technology has made the size and cost of digital systems practical for tasks deemed impractical a few years ago; however, software technology for the small computers used in digital weapons has lagged far behind hardware development.

b. A new generation of High Order Languages (HOLs) has given weapon software designers the tools required to produce efficient computer programs while deriving the many benefits of programming in an HOL. The best of these HOLs are implemented in modular, retargeting compilers which encourage standardization and flexibility.

3. Requirements:

a. The Air Force Armament Laboratory (AFATL) has a need for a

standard High Order Language (HOL) to be used in the implementation of real-time digital systems in tactical weapons. This includes midcourse and terminal navigation and guidance, digital autopilots, digital fuzing, and digital stores management.

b. The HOL must meet the following requirements:

- (1) It must be one of the languages chosen as an Air Force approved HOL in accordance with AFSC Sup 1 to AFR 800-14.
- (2) It must have constructs such as IF-THEN-ELSE conditional jumps, WHILE loops, variable declarations, and other such state of the art capabilities which support structured programming.
- (3) It must be well suited to the real-time applications required in AFATL.

c. The compiler implementation of the HOL must meet the following requirements:

- (1) It must be modular to allow the economic retargeting to new computers by replacing the code generator in an existing compiler.
- (2) It must use a high level intermediate language such as QUAD language as the primary communication between the front end and the code generator.
- (3) It must have a thoroughly documented interface between the front end and the code generator to simplify the task of writing code generators.
- (4) It must employ syntax optimization in the front end and target machine optimization in the code generator to assure maximum efficiency overall.
- (5) It must have the capability to output from one host computer to any of several target computers to end the present requirement of purchasing a software development station for each new imbedded type computer. This capability will also end the requirement to redo a software package each time a system must be moved to a new computer.
- (6) The compiler must be hosted on a computer that is accessible remotely via a dial up telephone network.
- (7) The compiler must be owned and controlled by the government and therefore not subject to license fees or uncontrolled modifications.

4. Approach:

a. Implement a JOVIAL J73 retargeting compiler which will meet the above requirements.

b. Study intermediate languages, concentrating on the following characteristics:

(1) Front end versus back end optimization.

(2) Ability to expand to accommodate new technology.

(3) The ability of the language to translate easily into the machine languages of computers with varying architectures.

c. Represent the interests of the digital weapon technology in the definition and implementation of the DOD common language.

5. Impact: This program will enable AFATL to develop JOVIAL J73 into a standard HOL for tactical weapon software. It will enable the Lab to remain abreast of any new developments in computer software technology, and it will give the Lab direct input to the DOD common language. This comprehensive program will put AFATL to the forefront in software technology and will maintain that position in the future.

What Will the Impacts
a Common High Order Programming Language Be?

Rob Kling and Walter Scacchi

Department of Information and Computer Science
University of California, Irvine
Irvine Ca. 92717

June 20, 1978

A common high order defense language has been proposed to help diminish the Babel of programming languages now used for embedded applications and to help diminish the maintenance costs of software projects [Fisher, 1978]. Despite these ends, it is not clear what such a language might accomplish without a careful specification of the "problems" which it is to solve. This paper and a companion piece raise a number of issues that trouble us in reading and discussion about the environment in which DoD-1 is to be used and the problems to which it is a solution [Scacchi and Kling, 1978].

It is commonplace to observe that most programming is undertaken within specialized groups. These specialists produce software products for yet other programmers or computer users. Despite this mundane observation, programming language development is largely treated as a technical problem with a traditional engineering sensibility. Typically, a skilled language designer tries to abstract a set of language specifications which will improve some aspect of programming or program use (e.g., efficient compilation, variety of data types, simple algorithmic representations) and to synthesize a suitable language to provide them. Thus viewed, a new programming language is another "tool" which competes in a marketplace with existing languages for attention and use. While most language designers hope that their products will meet with widespread acceptance, casual attention is given to some language features which should make the product technically superior, and "marketing" is delayed until a suitable product is in hand.

Despite the technical advances in programming during the late 60's and early 70's, many problems of program construction, testing, and use have remained relatively unresolved (e.g., program validation, insuring high quality and accurate documentation). One attractive strategy has been to increase the versatility of programming tools (e.g., with editors and debugging aids). Programming tools are claimed to be useful for developing of higher quality software. Many programmers prefer enriched programming environments. There is even some anecdotal evidence that they help speed the development of high quality software. But there is no systematic quantitative evidence that these enriched environments really pay off. And there is also some evidence that enriched environments can work against a programmer [Palme, 1978]. This phenomenon is rarely examined and poorly understood.

Most software designs are based on the belief that if one develops the technology "on its own terms," superior and usable products will result. According to this view, careful attention need not be paid to the social settings in which a given technology will be used. Such beliefs, when casually applied, lead to poor system implementations that do not have their intended effects, or which have unpleasant or unexpected social side effects. We believe this view is based on serious misunderstandings of the role of technologies in organizations. This paper sketches this argument in brief. Examples from programming and the DoD-1 development project are expanded in companion papers [Kling and Scacchi, 79; Scacchi and Kling, 78].

If one backs away from computing (for a moment) and looks at a wide array of technologies used in different social settings, one finds some interesting cases in which tools "developed on their own terms" had unexpected, negative effects when they were introduced in a given social setting:

1. In the late 60's GM built a new Vega plant in Lordstown, Ohio. During the previous decades, increasing automation on assembly lines had led to lower unit costs for cars, and GM's plant designers went all out to make Lordstown the most automated plant in the automobile industry. GM engineers claimed that the assembly line was "the fastest in the world." Technologists and economists could comfortably predict that the unit costs for Vegas would be lower at Lordstown than at more conventionally designed plants.

Oddly, workers at Lordstown found the pace of assembly too demanding and the associated rigid working conditions intolerable. Absenteeism and turnover were high. Cars were sabotaged by means such as placing loose bolts in the rocker panels. The Lordstown designers neglected the extent to which auto workers valued some control over the pace and conditions of their work [Rothschild, 1974].

2. "From the Phillipines to the Barbados, new mothers have been shunning breast feeding in favor of powdered formulas, even though they have no clean water for mixing them, no fuel to boil bottles and nipples, and no sanitary storage facilities. The babies do not get nourishment and immunities of their mothers milk, and the unsterilized and often diluted formula exacerbates the malnutrition and diarrhea that are chronic in the Third World."

"Some mothers reportedly stretch a four-day supply to four weeks. Others have substituted corn starch or cocoa for the formula, and one Nigerian woman used plain water in the belief that it was the bottle and the nipple that provided the nourishment" [Toth, 1978].

3. In the mid-60's, HEW funded a major municipality, Riverville, to build an automated information system (UMIS) to help "integrate services" between the 170 different welfare agencies providing services to its citizens and to increase the efficiency with which services were provided. Careful studies a decade later showed that UMIS was in place and is used routinely for recording the transactions between some agencies in Riverville and their clientele. However, the UMIS did not help integrate the operations of any agencies and is not used to increase administrative efficiency. Rather, the reports are used to help generate more Federal funding since HEW auditors believe that an agency with such extensive automation must be well managed, and that computerized data is more credible than manually aggregated data [Kling, 1978b].

The crux of these examples is that the "solutions" were defined to fit

narrow conceptions (or specifications) of the problems experienced by the technology users. In each case, the technical fix required more than simply "sticking" some technology in place. Various morals can be drawn from these examples. One can bemoan the decreasing industry of American workers or the intransigent featherbedding of public administrators. And one can hope that some mix of "education" and character reform will finally do the trick.

A more constructive approach views these examples from the vantage point of the people who were using each technology and notices the rationality of their actions. More deeply, technologies are used by people who are living and working in some social system which helps define the demands they face, their opportunities, and their beliefs about efficacious actions. Thus, the way a technology is actually used and its effects are very sensitive to the social setting in which it is used [Kling 1978a, 1978b]. It is true of tractors, baby's formulas, UMIS, structured programming, PL/1, FORTRAN, and DoD-1.

There are clear lessons from these examples for the DoD-1 development. In each of the examples cited above, the "problem" to be solved was defined in terms that were relatively insensitive to the social setting in which the technology would be used. Thus, "lower unit costs" and "increased services integration" were stripped out of their social context, converted into interesting, but underspecified technical problems, and then "solved" with products whose characteristics did not help solve the "real" problems. The technical fixes* did not create the desired social engineering because so little in the problem specification said much about the social setting. In fact, emphasis upon "the tool" tends to deflect attention from the characteristics of a social setting which may strongly influence the tool's use. And if using a tool introduces externalities (i.e., costs borne by people who do not receive the benefits of tool use), the tool metaphor does not help us understand them very well. Externalities are simply treated as a new constraint which an altered tool will meet, or a problem which another tool will help solve.

Alternative approaches are possible. Volvo, for example, decided to reorganize the work setting of automobile fabrication from single assembly lines to with static workers to work groups that "follow" a car through the stages of production. In the Volvo setting, it appears that workers are a little less efficient, but morale is higher, absenteeism and turnover are much lower, sabotage is not an issue, and the overall economies appear beneficial.

* Technical fixes are attractive because they enable one to focus on designing technologies which can be high spirited fun rather than upon the human dilemmas which can be woefully depressing. It's more fun to build a real-time information system that tracks lots of records than it is to study the routine demands of administrators in public agencies. It's fun to build powerful things that go fast, pull large loads, streak information from here to there, and it's consistent with our engineering skills. We shouldn't confuse our predilections for building such tools with solving anybody's real problems [Daedalus, 1977]. People who respond only to technical requirements can be stimulated to generate technical fixes [Dijkstra, 1978; Shaw, Hilfinger and Wulf, 1978]. A robust description of both the social and technological environments taken seriously would discourage technical fixes that may not work.

These approaches which pay attention to the context of tool use are not merely examples of "human factors" retrofitted onto a common technical core [Papenek and Hennessey, 1977] any more than making a graduate text on quantum electronics accessible to a lay public means simply designing a new cover and writing a new preface. Most of the interesting examples of "socio-technical" designs are now being carried out in industrial factories [O'Toole, 1974] or Third World countries [Papenek, 1973]. We have relatively few examples in computing, and little understanding about what is involved.

Let's again return to the case of software development. Certain commonplace "facts" should be underlined to illustrate the potency of social elements in software design:

1. JOVIAL, CMS-2, SPL-1, and TACPOL are dominant high-order programming languages used for developing embedded systems applications. Regardless how much one may prefer APL, PASCAL, or an extensible language, new languages that are technically preferable have not been accepted rapidly by industry programmers or their managers.
2. When a new software system is introduced, users profit from tutorial manuals to learn what the software system is good for and how to use it. Reference manuals are useful to learn or remember special features of a system. Those people who must maintain the system can use a good system design document. But, it is rare for a software application to have up-to-date, high-quality documents of all three kinds. One may attribute the common poor state of documentation to the absence of automated aids. But it also the case that preparing and updating suitable documentation is a kind of dirty work which many programmers and analysts prefer to avoid and for which there are few rewards in the computing world.
3. The extent to which programs are systematically tested before being placed into use varies considerably in the computing world. In fact, "testing" for some groups entails putting a program into a production setting, and then fixing bugs as they occur. Is it likely that system testing styles depend upon the organizational demands placed upon programmers rather than with their personality, IQ, or other individual characteristics of the programmers?

We have learned a great deal from empirical studies of program development. When Don Knuth rifled garbage cans at Stanford for FORTRAN listings, and carefully examined the kinds of statements and arithmetic expressions they contained, many were surprised by his findings. Similarly, when Barry Boehm and others studied the life cycle costs of software and found that maintenance dominated coding, software engineers took note [Boehm, 1972]. However, these studies are simple "peeks" into the world of software development. They help identify problem areas, or shift attention from one problem to another. But they only begin to explain why certain patterns recur. Thus, we know that 50%-90% of a program's life cycle costs can be from

maintenance, but we don't know much about the conditions under which maintenance is performed and how that alters costs. Current conventional wisdom is that well-structured code is easier to read and understand than "spaghetti coding," and thus that coding during development can influence the costs of maintenance.

It is, however, remarkably difficult to find good data about how programmers "doing maintenance" spend their time (and thus money). For example, how much maintenance time is simply spent deciding what changes should be made in negotiation with users? How much time is spent deciding where a program must be altered? If coding costs occupy a small fraction of development costs, why would one assume that they occupy a large fraction of maintenance time? It may be, for example, that maintenance costs are also high because:

1. Software design and implementation documents are of poor quality or are not kept up-to-date. Thus, programmers spend inordinate amounts of time figuring out how a piece of code works.
2. Maintenance programmers are frequently rotated. Soon after a person learns a system, he moves to another project.
3. Bugs are sometimes introduced by programmers who to make work for themselves (or others). Some cases of this have been reported.

Alternative explanations like these may best be resolved through careful empirical studies of programming practice. After all, different patterns lead to different understandings about "what the software maintenance problem is" and how it may be most efficaciously resolved. Good data about the what goes on during "maintenance" and where time and costs go would be invaluable.

One looks in vain for such understandings contained within the DoD-1 documents [DoD, 1978; Fisher, 1978]. The primary "problem definition" seems to emphasize:

1. Embedded applications have been written in over 450 different programming language dialects and maintenance costs are compounded by the difficulty of finding skilled programmers for a given language.
2. maintenance costs may be decreased during program development by providing helpful software tools.

In addition, certain "ill effects" of the current situation are identified (such as excessive dependence on certain vendors and diversion from important tasks), but are explained almost exclusively in technical terms. In contrast, potential problems caused by the introduction of DoD-1 are either ignored, or implied to be easily solvable through unspecified forms of management control.

Some of these difficulties may, however, defy simple solution. Vendors, for example, have a tremendous stake in hooking clients into effectively exclusive contracts. Using special dialects is but one strategy. Custom-tailoring software is another [Kling and Gerson,

1977]. Without careful analysis, "solutions" like DoD-1 may provide vendors with new opportunities to hook DoD into exclusive contracts. More seriously, programming is undertaken in work settings in which people are focussed not only on the logical task at hand, but also in doing interesting or low stress work, in maintaining career mobility, etc. These work contingencies lead to styles of software development which can be troublesome for end users. Such arrangements cannot be assumed to have benign effects without prior understandings of the organizational settings where system development takes place [Kling and Scacchi, 1978, 1979].

We believe that accounts such as these are likely to lead to a DoD-1 design which is as misfit as large tractors in Pakistan or UMIS in Riverville. Clearer understandings of the work settings in which DoD-1 will be used to develop embedded systems applications are essential to help further specify the design of DoD-1 features, particularly the software tools that should accompany it. Simple assumptions about the environments of use (e.g., that they are like the advanced laboratories of DoD-1 developers) will probably lead to yet another clumsy language and no overall savings. We refer the reader to our companion papers for examples which examine programming language use in greater detail [Kling and Scacchi, 1979; Scacchi and Kling, 1978].

References

1. Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment", Datamation, 19(5):48-59, May 1973.
2. Boehm, B., McClean, R.K., Vefrig, D.B., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March, 1975.
3. Daedalus of the New Scientist. "Pure Technology" in Technology and Man's Future A. Teich (ed.) New York: St. Martin's Press 2nd. ed. 1977.
4. Department of Defense "Common Language Environment Requirements" (Pebbleman) HOLWG, May 15, 1978.
5. Dijkstra, E.W., "DoD-1: The Summing Up," SIGPLAN Notices, Vol. 13(7): 21-26, (1978)
6. Fisher, David "DoD's Common Programming Language Effort" Computer 11(3)(March):24-33, 1978.
7. Kling, Rob "Information Systems in Public Policy Making: Computer Technology and Organizational Arrangements" Telecommunications Policy, 2(1)(March 1978):22-32

8. Kling, Rob "Automated Welfare Client-tracking and Service Integration: The Political Economy of Computing" (to appear) Communications of the ACM 21(5) (June, 1978)
9. Kling, Rob and Elihu Gerson "The Social Dynamics of Technical Innovation in the Computing World" Symbolic Interaction. 1(1) (Fall):132-146, 1977.
10. Kling, R , and W. Scacchi "The Social Character of Instrumental Computer Use" Technical Report #110 Department of Information and Computer Science University of California, Irvine, Irvine, Ca., 1978.
11. Kling, R and W. Scacchi; "The DoD Common High Order Programming Language Effort (DoD-1): What Will the Impacts Be?" SIGPLAN Notices, (Feb., 1979)
12. O'Toole, James Work in America Cambridge, Mass: MIT Press 1974.
13. Palme, Jacob "How I Fought with Hardware and Software and Succeeded" Software Practice and Experience 8(1) (Jan.-Feb.):77-83, 1978.
14. Papanek, Victor Design for the Real World New York: Bantam Books, 1973.
15. Papenek, Victor and James Hennessey Why Things Don't Work Pantheon Books, New York 1977.
16. Rothschild, Emma Paradise Lost: The Decline of the Auto-Industrial Age New York:Vintage Books, 1974.
17. Scacchi, W. and Kling, R. "DoD's Common Programming Language Effort: The Work Environments of Embedded System Development" Position paper, Irvine Workshop on the Environment for DoD's Common Programming Language, University of Calif , Irvine (29 June 1978)
18. Shaw M., P Hilfinger and W A. Wulf, "TARTAN - Language Design for the Ironman Requirement: Reference Manual; Notes and Examples," SIGPLAN Notices Vol. 13(9), pp 36-75 (1978).
19. Toth, Robert "Fitting Technology to Need Held Critical in Third World," Los Angeles Times (June 18, 1978), Section 1, pp. 1-32.

DoD's Common Programming Language Effort:
The Work Environments of Embedded System Development

Walter Scacchi and Rob Kling

Dept. fo Information and Computer Science

University of California, Irvine
Irvine, California 92717

In this paper, we will briefly discuss the social features of programming environments. The discussion takes place in the context of the effort to develop a common programming language (DoD-1) for use in embedded systems. We illustrate why language designers must carefully understand the interplay between the social and technical arrangements of software systems in the organizational settings where they are developed, implemented, and maintained. We emphasize patterns of software development that can occur in projects using DoD-1 to implement other applications rather than focussing on the implementation of DoD-1 itself.

Is "Random" Social Process Rational?

Many computing problems (such as how to reduce the cost of software maintenance for embedded systems) are commonly viewed as technically separable from their embedded social environment. Consider the following excerpt from the keynote address at the recent International Software Engineering Conference [Hoare, 1978]:

What is wrong with these products** is not the skill with which they are put together - that one can only admire! It is just the grotesque inadequacy of the original designs, which have emerged by an apparently random historical and political process, and pay not the slightest regard to the most elementary concern for matching technique to objective and objective to technique, for minimising cost and maximising benefit, in short for serving the use and convenience of man. (emphasis added)

While the accuracy of this perception of system quality is suitable, it is an inaccurate diagnosis of the problem. This diagnosis -- that designs emerge from random social processes -- shifts attention away from the intrinsic social interactions that take place in any system development effort. By underestimating the importance of the social processes that shape system development projects, we increase the possibility for prescribing unsuitable solutions.

From experience and limited observation of software system development projects, we know that determination of objectives, selection of techniques, the criteria used to assess and assign costs and benefits, are all interactive products of both formal and informal interpersonal (social) processes. While their actual sequencing may be complex and difficult to track, the interacting participants believe their actions are often quite rational, not random. The actions available to the participants are rational in terms of the

* the subject of the paper is whether software engineering is in fact an engineering discipline. Software engineering products are being contrasted to products of the other "traditional" engineering disciplines.

the author's examples include "...ships that will hardly stay afloat, planes that will hardly fly, schools that act as solar heat traps, and roads which cause traffic congestion." It is worth noting that these are all respectively embedded systems of one flavor or another.

demands and opportunities constraining them.

Organizational politics have their own rationality. In a recent study of computer acquisition and development decisions made in a large corporation, Pettigrew [1973] found that staff members who had the best "access" (e.g. trust of) higher level decision-makers, who could approve major computing decisions, had their way. When alternative technical strategies were all risky, members of the board of directors were likely to select the proposals of staff they knew and trusted. The actual technical merit of their proposals was not at issue. Sometimes their proposals were technically superior to alternatives posed by others, sometimes they were not.

These observations suggest that many social interactions which occur in computing settings (such as deciding which programming language to use) have a "local rationality." Such rationalities may not be apparent in (assumed) global perceptions of common computing environments. Moreover, hoping to change the calculus of local rationality by providing new tools depends as much upon knowledge of the constraints faced by the tool users as it does of the features of the tools.

The Software Fable

Current software tools (such as text editors, programming languages, and test data generators) have not been empirically or systematically demonstrated to reduce the cost of computing. Rather they have reduced the cost of access to and control over computation and computing system components (i.e., hardware and software). Many "myths" and sympathies exist which suggest that such software tools will reduce the cost of software development, use, and maintenance. But until empirical investigations are undertaken, the cost-reducing effectiveness of such tools remains an open research question, not a reliable fact.

Are Computing Resources Renewable?

Computing resources (such as CPU cycles) are often treated as renewable and extendable -- not scarce. This reinforces the emphasis on development. Maintenance, on the other hand, may be viewed as being oriented toward a conserving or nonrenewable resource perspective. Reductions in the cost of raw computing hardware act as an incentive to use more of the available raw resource because it is renewable. The preferred career and work contingencies for computing specialists to pursue are those directed toward system development. In addition, system development is usually defined by specialists as "creative" or "challenging" work. All this seems to exacerbate software maintenance since it provides incentives for more code to be developed that then must be maintained. Hence, software system life-cycle costs will continue to spiral upward.

Can New Computing Resources Satisfy Existing Demands?

The rate of demand for computational resources to support software systems outstrips the rates of availability of computing resources in most if not all computing environments. That is, as the cost of access to computational resources appears reduced through the availability of new tools and devices, the number of users increases, the number of new applications increase, and older systems are found to be increasingly procrustean. As system developers and users acquire greater skills and expertise vis-a-vis a system and computing, they often come to expect more from existing systems and from computing in general. These demands arise through experiences with computing and interaction among the participants in a setting.

Common organizational strategies for meeting increasing demands for computing rely on incremental adoption of additional or new resources. The availability of these resources, as noted above, is continually met with greater demands for more computing support. However, we note the time frame in which this occurs depends upon the dynamics of the particular computing setting. Thus, we should not be surprised to see that new technological innovations (e.g., faster, smaller hardware packages) continue to shrink the relative percentage of total computing system life-cycle cost reducible with such "fixes."

The Social Environment of Software Maintenance Work

Currently, we see a pervasive emphasis on software development over software maintenance.* Most of the research in software engineering (e.g., improving software reliability) is addressed to the development of new systems. The problem is that the effectiveness of logistical support and the timely operation of any embedded system critically depend on maintenance. The "attractive" assumption on which current software practices are based is that good development practices can significantly reduce maintenance costs. However, empirical research on the technical and social arrangements for software system maintenance is scarce.

Many computing specialists and programmers share a perspective in their definition of what work they prefer. To many specialists, development means design, build, test, document, and turn over to the user. The problem is that most of the work in programming is maintaining existing sometimes archaic software systems. However, maintenance entails on-going interaction with users, fellow programmers, managers, other people in the work environment, and the contracting (or supporting) organization. Current system life-cycle costs reflect the high cost of maintenance -- up to 90% of the total life-cycle cost of embedded systems [Fisher,78]. But typically, the cost figures do not reflect or distinguish costs between people's time, skills, budget, and inclination. However, we suspect these social resources are utilized in the interactions and negotiations

* Maintenance is often construed to be routine, non-stimulating work. Development on the other hand is said to be challenging and rewarding.

that transpire in determining what maintenance coding and documentation update work is to be performed.

A significant portion of research in software development is directed at developing tools to automate as much of the development process as possible. What happens if tools can be built which successfully automate much of these tasks? The concern here is for the potential conflicts that can surface when the automated or "routinized" production of software competes (on a cost basis) with the highly desired programmer task of software production. This scenario when pursued suggests that relatively fewer software programmers are involved in the automated production of software in contrast to the ever-growing armies of software maintainers [DeRoze and Nyman, 1978]. Thus we may see programmers resisting innovations in software development technology they perceive as a threat to their "bread and butter" work.

The Social Setting of Software Documentation

Adequate and up-to-date software documentation is continually a weak feature of most software systems. Poor documentation is not usually a result of software development practices. Rather, it is related to the software maintenance problem. Reasons for poor documentation are not so much due to the unavailability of suitable documentation support tools or deficient programmer practices. Rather updating documentation demands time and attention. Demands for a programmer's time and attention come from many different, contingent sources in the work environment. Software documentation upkeep is but one contingent demand. The notion of contingency is the suitable descriptor because with it, we can assume that there often exist conflicting demands for a programmer's time (staying on schedule), skill, money (keeping project costs from going over budget -- maintain economic efficiency), other opportunities (developing another new system or working on a proposal to land a new contract), and inclination (creative vs. routine work). It may well be reasonable to assume that a programmer will attempt to pursue a work style that is oriented toward satisfying those contingent demands which are most desired or rewarded.

On the Conversion of Existing Embedded Systems

Given the level of investment in software in embedded systems what "conversions" will likely occur upon the onset of DoD-1? Since DoD-1 is not intended to be used to convert existing systems, we won't worry about problems here -- yet. We can likewise divert discussion away from problems of converting or adapting current data sources. We can however, see that the advent of DoD-1 into the industries and organizations that build, use, and support embedded systems will have unclear impacts in their social environments.

We assume that large-scale training programs will be initiated and implemented for both entering and experienced specialists. Needless to say, those with the earliest, hence most experience will find new career options (including greater salary and mobility) opening for them. Career mobility can act to motivate a programmer desiring to acquire a position on software development projects -- by "staying out in front." Situations such as this manifest problems of retaining familiar and experienced programmers.

As these career contingencies become visible to larger numbers of programmers, they too may well attempt to follow suit. But what about the existing software systems? Who will maintain them? Who will want to have or keep the skills necessary to maintain procrustean systems written in archaic languages? If only development projects are to use DoD-1, then one way for a programmer to move from software maintenance work to development is to acquire expertise in DoD-1 thereby enhancing career opportunities. We note that with current embedded systems, personnel turnover is rapid, typically two years.

As expertise with DoD-1 becomes widely disseminated, finding programmers skilled at maintaining existing embedded systems -- written in one of 450 programming language dialects -- becomes more difficult, hence more expensive. These direct costs may be attributed to the social benefits accrued by individual programmers pursuing their career options.

When critical existing embedded systems become too costly or too unreliable to maintain then a reasonable rationale to follow would be to redesign and reimplement them. Many very large software systems consist of source code written years before by programmers no longer working with the system. Redeveloping such systems, however, implies that the existing systems are comprehensible with respect to their functions, implementation, and dynamics. Again, the availability of high-quality, up-to-date system design documentation would greatly facilitate system redevelopments (especially when the original developers are not available). Given that the appropriate documentation is unavailable or unsuitable, the redesign and reimplementation of an existing system will require greater effort than the initial system development.

While a new software tool like DoD-1 may be intended to be used only for the development of new systems, social forces within the current environments can create situations where "development of new systems" includes "conversion of old systems." Thus the life-cycle costs of embedded software systems continues to grow.

Computing and the Emergence of Problems

As computing continues to grow rapidly as a technology and as it continues to be applied to new application areas, we see that many unresolved socio-technical problems still exist. The prime example is building and maintaining large, reliable software systems. New sets of social and technical uncertainties continue to arise as computer

use permeates applied computational problem areas. This suggests that computing, as a social activity, may well be organized toward the production of problems in search of manufacturable solutions. As such, the implied production cycle will not converge: many problems will never reach complete solutions. This keeps computing alive as an interesting socio-technical arena but it suggests the cost of computing will continue to rise.

As Software Costs go up, Computing becomes Political

In the real world of organizations, budgets are finite and packed. Organizational demands for computing resources compete with other organizational needs and demands. Such contentions are resolved by non-random historical and political processes within and between organizations, their participants and representatives. These situations are characterized by policy acts, interpersonal bargaining, formal negotiation, "arm-twisting" and the like. These activities may be discussed in technical terms or as purely technical matters. But nonetheless, the politics of organizational resource distribution will influence the shape of computing arrangements. Thus as the material, labor, and social costs of computing continue to rise, computing will become more an object of contention when organizational resources are being allocated.

Tentative Conclusions

1. Technical and social aspects of the programming work environments shape software system development, use, and maintenance. As such, these topics and their related social and technical problems cannot be treated in isolation from each other. To do so can result in the development of tools, techniques, or other "solutions" that don't fit in computing environments and thus fall into disuse. Such an outcome would itself be costly.
2. When seeking effective means to temper the growing cost of software systems, it is necessary to consider both technical and social arrangements of the environments where the systems are intended to be built or operate.
3. Socio-technical processes affecting the increasing cost of software include: the nature of software maintenance, the nature of programming work and documentation upkeep, the organizational dynamics of the settings where software systems are used, and nature of converting "obsolete" systems.
4. Research and extended discussion on the interplay of social and technical exigencies in the selection and use of programming languages is lacking but needed. The success of DoD-1 hinges on social arrangements of programming environments which may not be commonplace. Systematic studies which identify the likely conditions of DoD-1 use would provide credible evidence for the

likelihood of it's success.

5. There are no easy or simple solutions at hand. Emphasizing the technical side of software -- through the creation of new software tools -- as the means to solve the high cost of software problem is short-sighted. Social patterns within software development groups may have as potent an effect on the quality of products produced as do the software tools used to implement them. Attending to social processes casually or late may well turn out to be insufficient.
6. While opportunities to influence the requirements of the DoD-1 programming language are gone, there is still time to consider and influence the social and technological contours of DoD-1 programming environments.

References

1. De Roze, B.C. and T.H. Nyman, "The Software Life-Cycle - A Management and Technological Challenge in the Department of Defense," IEEE Trans. Soft. Engr., Vol. SE-4(4): 309-318, (July, 1978)
2. Fisher, D.A.; "DoD's Common Programming Language Effort," Computer, 11(3), pp. 24-33, (March, 1978)
3. Hoare, C.A.R.; "Software Engineering: Keynote Address," Proc. Third Inter. Conf. Soft. Engr., IEEE Press, pp. 1-4, (May 1978)
4. Pettigrew, A.M.; The Politics of Organizational Decision-Making, Tavistock Press, London, (1973)

Assumptions About the
Social and Technical Character
of Production Programming Environments*

Rob Kling and Walt Scacchi

Dept. of Information and Computer Science
University of California, Irvine
Irvine, Calif. 92717

22 June 1978

* Based on a session held at the Irvine Workshop on the Environment Requirements for DoD's Common Programming Language. The discussants included Robert Anderson, David Fisher, Dennis Kibler, Duncan Morrill, Patricia Santoni, and the authors.

Background

Throughout the Workshop on the environment for DoD's common higher order programming language (DoD-1), participants in each session have often made different assumptions about the production setting in which the language would be used. In characterizing the size of programs to be written (.5K lines to 5000K lines), the size of host machines (32K bytes of memory to 1024K bytes of memory), the skill of programmers (highly trained in modern programming methods or high school graduates with 60 days of training), different participants have made different assumptions. Also, since it is hoped that DoD-1 will help decrease the costs of maintaining software, different assumptions about what "maintenance" entails, why it is so expensive, and how software tools can cut the costs are critical. Usually, Workshop participants have not made their assumptions explicit. Nor is there an easily accessible document which explicates a set of assumptions of the social and technological arrangements to which DoD-1 can be designed. We believe that a clear understanding of the social environments where DoD-1 is to be used is extremely critical in shaping a useful and usable programming language environment design. Otherwise, it may not fit well in the software production environments in which they were intended to be used.

Throughout the development of DoD-1, the basic problem to be considered is (how to reduce) the high cost of software. Though we now know that the dominant life-cycle cost of software occurs in software maintenance, what can we attribute these costs to? What is the public knowledge of software maintenance costs in the research, academic, industry, and military organizations? If we have no specific, general, or even weak common understandings of the social and technical dynamics of software maintenance in different work environments, how can we expect to develop tools and techniques which will effectively attack the high cost of software problem?

These questions and others [Kling and Scacchi, 78,79; Scacchi and Kling, 78] helped set the stage for a Workshop session on the interplay between the social and technical features of production programming environments and their dynamics.

Assumptions that Relate DoD-1 to Software Maintenance

In discussions about the design of DoD-1, the participants seemed to differ in five different areas:

1. What does "maintenance" entail? How does it arise? How can maintenance costs be cut?
2. Current Programming Tasks -- What is the size of the systems that DoD-1 will be used to fabricate? What kinds of applications are most likely to be developed? What is the role of the host computer (development machine vs. target machine)?

3. What skills can DoD-1 programmers be expected to have? How highly motivated will most of them be to learn modern software development methods, and to carefully document and test the systems they develop or maintain?
4. To what extent do project management constraints, styles of management control, or manager's ideology influence the approaches to software development followed on particular application projects?
5. How well will DoD-1 fit into existing programming environments? Do the DoD-1 developers (and their supporters) expect to use DoD-1 as a means to bring about changes in current programming practices (i.e., DoD-1 as a "Trojan Horse")?

A programming language development effort like DoD-1 cannot be expected to produce a language that is "everything for everyone." As such, many choices and decisions must be made to decide which features or attributes the resulting language will possess. We see that many such decisions are influenced by the alternative (real or conjectured) programming environments one uses for reference.

Our discussion now turns to elaborating what is known about the current production programming environments.

Production Programming Environments made Explicit

To help establish a context for viewing programming environments, it was noted that DoD-1 is targeted for:

1. Development of new embedded software systems
2. Development of software tools (e.g., to provide tie-ins for tools to be made available in common software libraries in a standard language).

We now make the assumptions about the programming environments where DoD-1 is intended to be used explicitly (i.e., "what it's like out there in the trenches")*.

Maintenance is considered to be everything that happens to software after user acceptance. Maintenance can range from "bug fixes" through the complete redesign and development of a delivered system. With current embedded systems the people who develop the systems are often not the same as those who must maintain it. Software applications for DoD are often developed by industrial contractors who employ programmers with 2-4 years of college. Software maintenance is often performed within DoD installations by programmers with high school educations and little exposure to modern

* This account was abstracted from discussions with the acknowledged participants. It is not based on systematic empirical studies of military and industrial software development environments.

software development methods.

While it is true that software must be alterable, with current embedded systems, the number of lines of code changed in a system annually (throughout its 10-25 year life-cycle) is on the order of the number of lines of code in the system.

Most embedded military software systems are large: between 50,000 and 2,000,000 lines of code. Most computers used in these applications are relatively obsolete (most software is written for computers procured as long as 15 years ago). Despite the existence of military standards that require the use of high-level languages (i.e., large fractions of those programs to be written in high level source code), the use of low-level languages often dominates for cost reasons ("it's too expensive to develop new tools, when project deadlines must be met within budget"). In fact, it seems that subversion of the existing high-level standard is claimed to be commonplace. One explanation is that existing languages (e.g., TACPOL, CMS-2, JOVIAL) do not have appropriate features for handling parallel computation, real-time clocks, arbitrary I/O devices, interrupts, etc. According to this line of reasoning, if DoD-1 provides a sufficiently rich array of features, it will be "naturally" attractive.

Many software production environments only have software tools that were available 10-15 years ago together with programmers primarily skilled in assembly language programming. Thus many people believe that the introduction of almost any modern software development tools (e.g., library support, cross-reference aids) could not help but improve the quality of typical software development efforts. (Note: The DoD-1 specification allows programmers sufficiently flexible use of GOTO's and assembly code linkage that one can literally write FORTRAN or assembler in DoD-1.)

The pivotal individuals whose choice of commitment to adopt or resist the introduction of DoD-1 are those in middle management positions in current software development environments. Staff retraining costs will largely come from their budgets, but they will not see the savings of lower software maintenance costs. Thus, under present funding arrangements, they have little incentive to invest heavily in adopting modern programming methods as an adjunct to DoD-1 use. Since project funding limitations are real, since production schedules must (or should) be met, and since programmer (re)training takes time and money, these managers will decide if adoption of DoD-1 is cost-effective in their situations. It is difficult to predict the extent to which many middle managers will seriously adopt DoD-1. The decision to operationally adopt DoD-1 is not merely one of choosing a "better" technology. This choice cannot be mandated unless resource support is made available.

Large-scale commitment to DoD-1 is likely to be evolutionary: gradual and fiscally tempered. But the repercussions of this observation for language design, programming environment development, or training programs are unclear.

There is consensus that many applications written in DoD-1 for small target machines will be developed on larger machines and then cross-compiled. Some even assume that separate machines (e.g., host and target) will be the dominant mode of DoD-1 use. Use of DoD-1 in the development and maintenance of new embedded software systems will require large, resource-sharing computing facilities (i.e., anywhere from larger minicomputer systems up to the likes of the National Software Works and the ARPANET). Such facilities may not be available or accessible under current arrangements to most software development groups.

Contractors differ substantially in the ease with which their project staff can gain access to large machines. The best sites have easy access to a wide array of facilities. But there are many lesser sites where good access cannot be taken for granted. Such facilities, though, must be accessible to the average (mediocre) programmer given his level of skills, available time, inclination, and organizational commitment of resources (training and computing budget) to ensure use.

Summary and Conclusions

1. Since large applications will be developed with DoD-1, there should be an early commitment to developing integration aids (e.g., libraries, cross-referencing packages). These may have to be developed with external support rather than waiting for vendors or user groups to develop them in the next decade.
2. It appears that many applications will be developed on a large host machine and then cross-compiled to a smaller target machine. This pattern makes sense from the point of view of providing rich software tool environments for DoD-1, but also makes large demands for new computational resources. The extent of these demands and how they will be met is still unclear.
3. It is tempting to take a unified view of minor alterations to programs by treating "fixes" as a form of redesign. According to this view, all alterations should be referenced back to the original specification and treated coherently as redesign through all levels of documentation and coding. This view has an attractive intellectual coherence and may make good practical sense for programs of 100 lines. But its applicability to programs of 1K, or 50K, or 5000K lines merits careful investigation.

There may be other approaches to program development which make interesting demands on both programming style and the DoD-1 environment which merit careful scrutiny when applied to massive programs.

4. DoD-1 programmers cannot be expected to be highly skilled. The language -- including its software tools and teaching aids -- will have to be targeted towards the "middle" 50%-80% of programmers in order to be effectively received.
5. There are relatively few incentives for many "middle managers" to aggressively embrace DoD-1 and its associated programming methodologies.
6. "Evolutionary" utilization of DoD-1 may well mean that many DoD-1 programs will not rely upon elegant control structures in the language. Instead, we may expect to find FORTRAN-like programs in DoD-1. Systems may well evolve in "mixed" modes. Differing levels of programming sophistication and culture may appear in different portions of a given large application.
7. Maintenance is still poorly understood since it seems to cover everything from bug fixes, through enhancements, to the complete redesign and redevelopment of systems. There is relatively little understanding of the programming development environments in which modern software methods are being used. And there is little understanding of the differences between those environments that are rich in resources and highly professionalized versus those that are less so.

References

1. Kling, R. and W. Scacchi; "What will the Actual Impacts of a Common Higher Order Programming Language Be?", Position Paper, Irvine Workshop on the Environment for DoD's Common Programming Language, University of Calif., Irvine (June 20, 1978)
2. Kling, R. and W. Scacchi; "The DoD Common High Order Programming Language Effort (DoD-1): What Will the Impacts Be?" SIGPLAN Notices, (Feb., 1979)
3. Scacchi, W. and R. Kling; "DoD's Common Programming Language Effort: The Work Environments of Embedded System Development," Position Paper, Irvine Workshop on the Environment for DoD's Common Programming Language, University of Calif., Irvine (June 20, 1978)

A Practical, Precise, and Complete Standard Definition
for the DOD Common Programming Language

Eldred Nelson
TRW Defense and Space Systems Group

Introduction

As noted in the preliminary Environmental Requirements document¹, the DOD common language needs a standard definition specifying the syntax and semantics of the language and providing a basis for deciding whether or not compilers conform to the definition. Such a standard definition is fundamental to achieving the objective of the new common language - a single programming language, usable for programming all embedded computer systems, having programming manuals consistent with compiler implementations, and having consistent compiler implementations across a wide family of computers.

SEMANOL is a formal language standardization technology, capable of providing the needed standard definition. It is proven through application to JOVIAL (J3), JOVIAL (J73), UCMS-2, and BASIC. The SEMANOL system is based on a mathematical theory of semantics, providing a sound basis for its application. SEMANOL is a syntactic and semantic definition language, designed to be relatively readable by people and executable by computer, through a SEMANOL Interpreter. With SEMANOL, a programming language can be specified precisely and completely - context-free syntax, non-context-free syntax, input-output semantics, operational semantics, and implementation dependencies. The detail in a SEMANOL specification can be controlled to a degree defined by the language control authority, so as to constrain compiler writers enough to produce uniform implementations but without unduly constraining their efforts to produce efficient code. Machine dependencies in the language are specified in a parameterized way, allowing comparison of semantic effects in different machine environments and permitting machine dependent semantics to be limited. Because the SEMANOL specification is executable, test cases can be developed to test the language definition to a measured effectiveness while minimizing test case redundancy. These test cases serve to thoroughly debug the language specification and to provide a test of compiler conformance to the specification. Semi-automated test case generating tools using SEMANOL are under development.

Both of the language design contractors have indicated, in their preliminary design documentation, plans to prepare an axiomatic formal specification. A SEMANOL specification will complement the axiomatic specification, providing the missing non-context-free syntax and semantics of execution.

A standard definition of the DOD common language, suitable for use by the Configuration Control Board and Language Support Agency, would comprise:

- a formal SEMANOL specification of the language,
- an English reference manual, prepared using the SEMANOL specification as a basis for writing accurate and complete descriptions of language features,
- an axiomatic specification suitable for use in formal verification,
- a set of test cases, thoroughly exercising the SEMANOL specification, for validating compiler conformance to the specification.

With such a definition, compilers can be produced implementing the language uniformly, supporting greatly increased transportability of application software and software tools.

In following sections of this paper, the language definition problem is discussed; the SEMANOL system and its underlying mathematical theory of semantics are described; application of SEMANOL to the DOD common language is outlined; how the SEMANOL specification is used to generate test cases for demonstrating compiler conformance to the specification is described; the relation of a SEMANOL specification to an axiomatic definition is discussed; a formal standard definition for the DOD common language is described; and use of the standard definition to support language control is discussed.

The Language Definition Problem

Having a standard programming language and reaping the benefits of the standardization are dependent on having a standard definition usable in distinguishing a standard implementation from a variant and in assuring

that an implementation does indeed meet the standard definition. Standardization efforts for COBOL, FORTRAN, and other languages lacking such a standard definition have had only a limited success.

Conventional programming language specifications, written primarily in English prose, are known to be imprecise and incomplete. They contain ambiguities and in some cases are internally inconsistent. These deficiencies create problems for compiler writers and programmers. Lacking a precise and complete specification, compiler writers must invent a portion of the language. This invented portion of the language is generally not documented, so programmers must discover it by trial and error, increasing the cost of debugging programs.

Recognizing these problems, the IRONMAN² general design criterion 1H specifies that "To the extent that a formal definition assists in achieving the above goals, the language shall be formally defined." The contractors for both the red and green languages indicated in their preliminary design documents they intend to provide an axiomatic formal definition as part of the detailed design.

The axiomatic formal definition is an important step towards a standard definition, providing axioms defining data structures, control structures, and assignment; however, all axiomatic language specifications prepared to date are incomplete. They generally assume that a context-free-syntax is defined by some other method, make no effort to deal with non-context-free syntax, and omit significant execution details and implementation dependencies.

The SEMANOL System

SEMANOL was developed to provide precise and complete specifications of programming languages, comprehensible to a suitably indoctrinated reader; i.e., SEMANOL is designed to supply people with a basis for communication about programming languages more precise and complete than commonly employed description methods.

SEMANOL is based on a mathematical theory of the semantics of programming languages³, which represents a programming language as a system of 5 components: (P,I,F,T,Φ).

P is the set of executable programs in the language. I is the set of input and output values for these programs. F is the set of computable functions specified by the programs in P . T is a set of execution traces - ordered records of semantic actions in executing programs. Φ is a semantic operator defining how a program in P computes the values of a function in F .

In the semantic theory, a program $p \in P$ specifies a computable function $f \in F$ on a set $E \subseteq I$. Execution of p with an input $E_1 \in E$ computes the function value $f(E_1) \in I$. The execution trace $t \in T$ defines the semantic actions in computing $f(E_1)$. The SEMANOL system formal definition of a programming language L is:

$$L = \{(p, E, f) : \text{FOR ALL } E_1 \in E (\Phi(p, E_1) = (t, f(E_1)))\}$$

i.e., L is the set of triples (p, E, f) such that for all inputs E_1 in the input domain of program p , the semantic operator Φ applied to p and an input $E_1 \in E$ constructs the function value $f(E_1)$ and the execution trace t . This definition specifies input-output semantics by associating each program p with an input domain E and a function f on E . It specifies operational semantics through the execution trace t , defining how $f(E_1)$ is computed from p and E .

The semantic operator Φ is a function from $P \times I$ to $T \times F$. It specifies the set P of programs in terms of:

- constants
- variables,
- expressions formable from strings of constants, variables, defined terms, and quantifiers, and
- restrictions (non-context-free) on the scope of constants, variables, definitions, and expressions;

it specifies the set I of input and output values; and it specifies the execution of programs in terms of expression evaluation rules, execution sequencing rules, and the effect of machine dependencies (in parameterized form). Thus Φ specifies both the syntax and semantics of L .

Semantic operators for programming languages are constructed as programs in SEMANOL (SEMANTics Oriented Language), a language developed for that purpose. Because SEMANOL is a programming language, it, too, has a formal definition in

terms of a semantic operator. Such a semantic operator for SEMANOL has been constructed in the form of a SEMANOL Interpreter, operational on the HIS 6180 computer under MULTICS at Rome Air Development Center (RADC), and accessible via the ARPANET. An informal description of SEMANOL is provided in the SEMANOL(76) Reference Manual.

The SEMANOL Interpreter⁴ is constructed in two components: (1) a SEMANOL Translator which translates a SEMANOL specification of a programming language into an internal form, SIL (SEMAMOL Internal Language) and (2) an EXECUTER which, using the SIL representation of the language specification, parses programs in the specified programming language, applies non-context-free tests, interprets SEMANOL operators and constants, produces the correct output of program execution, and provides a trace of program execution, which can be selected as a user option. Figure 1 shows how the SEMANOL system would function, applied to the DOD common programming language (designated DOD-1 in the diagram).

The SEMANOL Metaprogramming Language

SEMAMOL is a "metaprogramming" language specifically designed for programming semantic operators. Its design stresses readability. It has high level expressiveness and uses conventional notation where possible. SEMAMOL has evolved as experience in applying it to specifying programming languages - JOVIAL (J3)⁵, JOVIAL (J73)⁶, UCMS-2⁷, and BASIC⁸ - has shown which expressions and terms are useful and readable. It has a standard version, SEMAMOL(76), defined by a SEMAMOL(76) Reference Manual⁹, with the SEMAMOL Interpreter providing executable confirmation of interpretation of SEMAMOL expressions. SEMAMOL(76) primitives were chosen to have a direct correspondence with well understood mathematical objects (e.g., #INTEGER) and operators (e.g., +). The Reference Manual defines many of the high level expressions in terms of SEMAMOL primitives, using SEMAMOL.

How SEMAMOL is used to specify a programming language may be seen in terms of the specification prepared for JOVIAL (J3). The SEMAMOL program for the JOVIAL (J3) semantic operator comprises four sections:

- Declarations Section, declaring global variables,
- Context-Free-Syntax Section, defining a context-free-syntax in terms of a grammar and a Lexical Syntax,
- Control-Commands Section, defining execution of the SEMAMOL program,

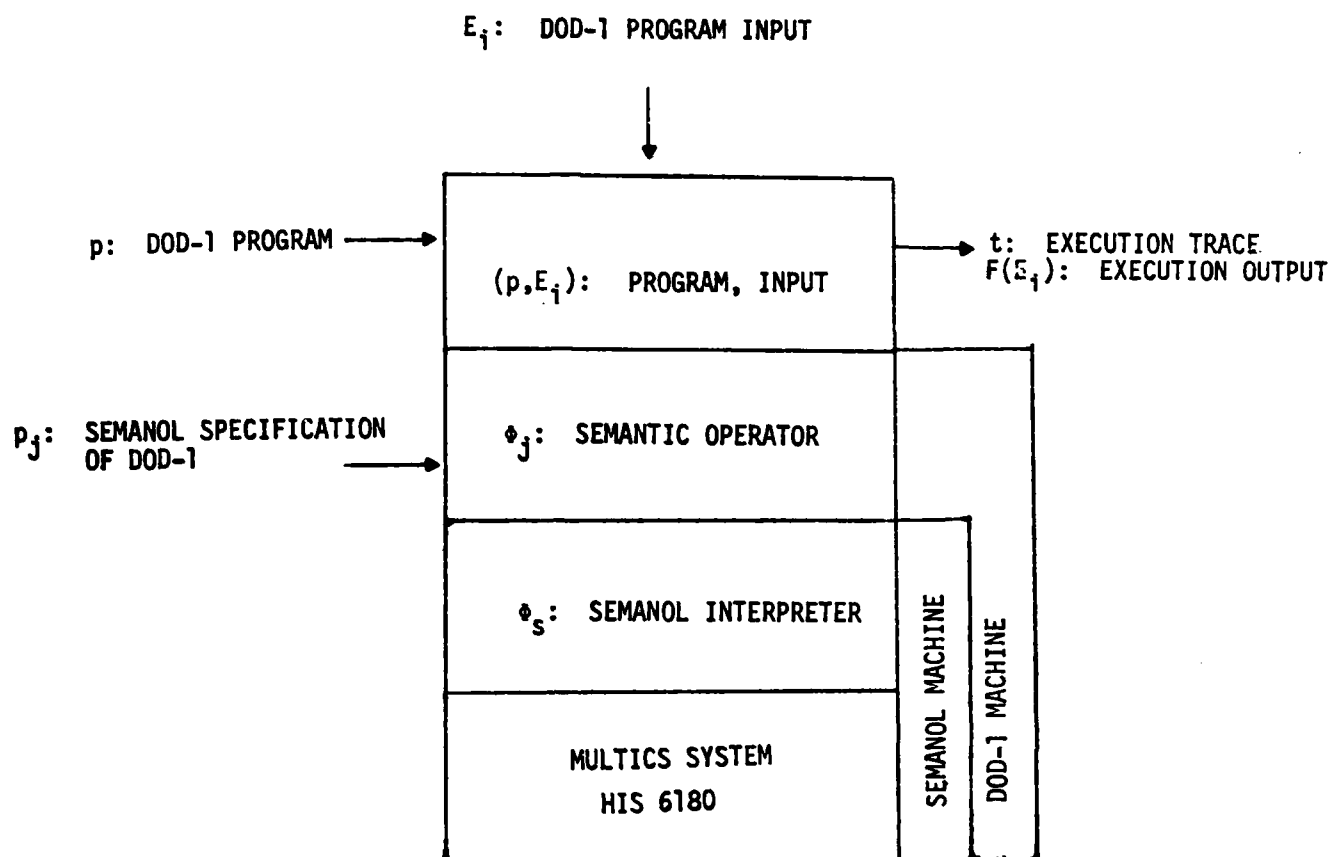


FIGURE 1: THE SEMANOL SYSTEM

- Semantic Definitions Section, defining Lexical Analysis, Context-Sensitive Checks, Control Semantics, Evaluation Units, Evaluation Semantics, Type Definitions, Semantic Attributes, Selectors, Implementation Parameters, Auxiliary Definitions, Scoping Contexts, Names, Generalized Assignment, Standard Reference Addresses, Addressing Units, Addressing Unit Addresses, and Relative Addresses.

The Declarations Section declares five global variables used in the SEMANOL program. Because it is short the entire section is reproduced here:

```
#DECLARE-GLOBAL:
    current-executable-unit,
    jovial-system,
    ncf-error-is-discovered,
    transformed-token-seq,
    unscanned-token-seq #.
```

"#DECLARE-GLOBAL:" is a SEMANOL keyword denoting that the following strings are global variables. "#." is a SEMANOL symbol denoting the termination of a SEMANOL statement. Most SEMANOL keywords have the symbol "#" as their first character. The five global variables declared have the syntax of SEMANOL names and are constructed from English words suggesting the sets over which the variables may range.

The Context-Free-Syntax Section defines syntax in a form similar to BNF notation, as this example shows:

```
#DF jovial-j3-system
=> <gap><jovial-j3-program><gap>
    <optional-library><optional-compools>
    <defaults> #.

#DF jovial-j3-program
=> <optional-control-input><program> #.

#DF optional-control-input
=> <#NIL> #U <implementation-control-input><gap> #.
```

```

, #DF implementation-control-input
=> <'compoools'><gap><':'><gap>
    <compool-name-list><gap><'$'> #.

```

The SEMANOL keyword "#DF" introducing each statement denotes that the statement is a definition statement. In the context of the Context-Free-Syntax Section, #DF denotes a syntactic definition defining a syntactic class name for the set of strings specified on the right hand side of the "=>" symbol. Each pair of angle brackets, "<gap>" e.g., denotes a set of strings and adjacent pairs of angle brackets denote the set of strings formed by concatenating strings chosen from each of the two sets. "#NIL" is a SEMANOL keyword denoting a string of no characters and "#U" is a SEMANOL operator denoting the union of the sets on its left and right. <'compoools'> denotes a singleton set whose member is the string "compoools".

The Control-Commands Section is a high level description of execution of the SEMANOL specification, using syntactic and semantic definitions in the other sections. Since the text of this section is relatively short it is reproduced in its entirety:

#CONTROL-COMMANDS:

```

#ASSIGN-VALUE! jovial-system = #CONTEXT-FREE-PARSE-TREE
(textually-transformed (#GIVEN-PROGRAM), "with-respect-to"
<jovial-j3-system>)

#IF ($jovial-system$)is-not-syntactically-valid
#THEN #COMPUTE! #ERROR
#IF there-are-executable-units-in (main-program-of
(jovial-system)) #THEN
#BEGIN

#ASSIGN-VALUE! current-executable-unit =
first-executable-unit-in-program
(main-program-of(jovial-system))

#WHILE ($current-executable-unit$)is-not-terminator #DO
#BEGIN

```



```

#COMPUTE! computational-effect-of
(current-executable-unit)
#ASSIGN-VALUE! current-executable-unit =
executable-unit-successor-of (current-executable-unit)

```

```
#END
```

```
#END
```

```
#COMPUTE! #STOP #.
```

This section directs that a context-free parse tree of the JOVIAL program be constructed, using the context-free-syntax specified in the Context-Free-Syntax Section. If the program is syntactically valid and contains executable units, execution begins with the first executable unit in the program and continues with the executable units successor until a terminator is reached. These processing steps are articulated more precisely in the control portion of the Semantic Definitions Section. The "with-respect-to" in the first statement is a comment inserted to aid reader interpretation of the statement.

After the context-free parse tree is constructed, context-sensitive checks are applied; e.g., in JOVIAL (J3), a return statement may appear only in a processing declaration. This is expressed in SEMANOL in terms of semantic definitions as:

```

#DF test-if-returns-are-all-in-proc-decl-of(prog)

=> #NIL #IF #FOR-ALL return-stmt #IN
sequence-of-returns-in(prog) #IT-IS-TRUE-THAT
(($return-stmt, "of" prog$) is-in-some-proc-decl);

=> error-message('RETURN-OCCURS-IN-MAIN-PROGRAM')
#OTHERWISE #.

#DF is-in-some-proc-decl(stmt, prog)

=> #THERE-EXISTS proc-decl #IN sequence-of-proc-
decl-in(prog) #SUCH THAT (($stmt, "in"
proc-decl $) occurs) #.

```

This test provides an error message if a return is found which is not in a processing declaration.

An example of control semantics, specifying execution sequencing is:

```
#DF statement-following(stmt)

=> ((next(stmt, "in" sequence-of-executable-
      statements-in(program-unit-containing(stmt)))) #.

#DF next(stmt, "in" seq)

=> ((#ORDPOSIT nx #IN seq) +1) #TH-ELEMENT-IN seq #.
```

An example of evaluation semantics is:

```
#DF product-value(unit)

=> integer-product(operand 1-of(unit), "*"
      operand 2-of(unit)) #IF type(unit) #EQW 'integer';

=> fixed-product(operand 1-of(unit), "*"
      operand 2-of(unit)) #IF type(unit) #EQW 'fixed';

=> floating-product(operand 1-of(unit), "*"
      operand 2-of(unit)) #IF type(unit) #EQW 'floating' #.
```

Here "unit" is a multiplication expression whose evaluation semantics depends upon the type of expression.

Implementation dependencies are expressed in terms of parameters, such as "bits-per-word", e.g:

```
#DF implementation-integer-subtract(x,y)

=> (($x$) converted-to-standard-form - ($y$)
      converted-to-standard-form $)
      with-result-converted-to-implementation-form #.
```

This definition defines implementation dependent subtraction of two integers (y from x) in terms of:

- converting each integer value to a standard form
- subtracting the integer values in the standard form
- converting the result to an implementation form.

The "(\$"and"\$)" denote that the value enclosed by the parentheses is the argument of the following function.

```
$DF with-result-converted-to-implementation-form(sem-const)
"{$sem-const$) is-semanol-base-2-integer-constant})"
```

```
=> ($ #PREFIX-OF-FIRST '#B2' #IN sem-const $)
conformed-to-implementation-word-size #IF
#FIRST-CHARACTER-IN(sem-const) #NEQW '-';

=> ($($($word-between('-', "and" '#B2', "in" sem-const)
$) conformed-to-implementation-word-size $)
complemented $) incremented-by-one #OTHERWISE #.
```

The standard form for an integer value is a SEMANOL base 2 integer constant of the form <sign><i><'#B2'>. For positive integers, sign is the nil string and for negative sign is the symbol "-". i denotes a string of 1's and 0's with the left hand bit having a value 1 (i.e., zeros are suppressed). "#B2" denotes that the string i is to be interpreted as a binary integer.

```
$DF conformed-to-implementation-word-size(val)
"{$val$) is-string-of-ones-and-zeros})"

=> #RIGHT bits-per-word #CHARACTERS-OF(($bits-per-
word $) zeros #CW val) #.
```

Since the value "val" is in standard form with its left zeros suppressed, conversion to implementation word size involves concatenating (\$CW) zeros on the left of val and selecting the right (bits-per-word) of that string of ones and zeros.

Application of SEMANOL to the DOD Common Language

Preparation of the SEMANOL specifications of JOVIAL (J3), JOVIAL (J73), UCMS-2, and BASIC have shown that it is feasible to prepare a formal specification defining the syntax and semantics of real programming languages, and that such a specification is relatively readable by people and executable on a computer. In the preparation of each of these formal specifications, a number of ambiguities and incomplete definitions were identified. The ambiguities were resolved and the definitions completed, confirming the now widely accepted notion that preparing a formal specification can aid language definition and standardization in this manner. However, the specification development also identified some language definition issues

important to DOD common language development. The non-context-free syntax and implementation dependent semantics, neglected in most formal specification approaches, are major portions of the language specifications.

Although much of the discussion concerning language design, in the literature and in design documents, considers syntactic issues primarily in terms of context-free-syntax, the real programming languages used in most application and system programming contain many non-context-free restrictions. These restrictions ranges from restrictions on the scope of variables to restrictions on how various constructs are used in specific contexts. The general design criterion for reliability of the DOD common language tends to increase the non-context-free syntax, for many of the techniques promoting the production of reliable programs involve language enforced restrictions, defined in non-context-free syntax. SEMANOL specification of the DOD common language will identify these non-context free aspects and describe them, so they may be understood and properly implemented.

In spite of the general design criterion of machine independence, the requirement that the DOD common language be designed to support software development for embedded computer systems will lead to a substantial amount of machine dependency, for embedded computer system software generally involves development of operating systems or modification of vendor supplied operating systems. This necessarily requires facilities for describing and managing machine resources, many of which are machine dependent. SEMANOL specification of the DOD common language will identify the machine dependencies, characterizing them, where possible, in parameterized form.

Identification of the machine dependencies during the design effort could help limit their effect. Most language definitions and implementations are more machine dependent than necessary. The definition document for JOVIAL (J3), e.g., attempting to be precise, describes computational effects in machine dependent terms. SEMANOL specification of semantics can illuminate such situations, showing how they can be defined in machine independent ways. Where machine dependence is inevitable - e.g., the effect of finite word length on multiplication and division - SEMANOL specification can be used to limit that dependence to the minimum necessary.

The discussion of machine dependence leads into consideration of the amount of detail in a formal specification. Compiler writers have been resistant to the use of formal language specifications, contending that formal specifications overconstrain the language implementation by specifying how things are done rather than just what is to be done. The SEMANOL specifications previously prepared, being executable and developed to demonstrate the precision and completeness of the technique, do constrain implementations more than necessary. Although a SEMANOL specification contains the operational semantics, it does not need to constrain compilation details. Instead of specifying an implementation exactly, SEMANOL can specify a set of alternate implementations, while retaining its executable nature by selecting one for execution with the SEMANOL Interpreter. In effect, parameterizing machine characteristics such as word length specifies a set of alternate implementations; however, some sets will require more complex specifications. An issue to be determined for the DOD common language is the degree of detail in the formal specification.

The DOD common language contains advanced features not previously described in formal specifications. Although some of them are relatively straightforward, others are more complex - e.g., facilities for parallel processing. These new features, which are expected to evolve during the design contractors' development effort, will require some study and analysis for their precise specification.

Testing Language Specifications and Compilers

Because the SEMANOL specification is executable, it can be tested using machine methods. This enables the SEMANOL specification to be debugged to a degree not feasible with non-executable specifications. In addition, it enables the development of test cases having a measured effectiveness. A SEMANOL specification, being a program, is composed of executable elements. Each test case will exercise a sequence of these executable elements, and the SEMANOL Interpreter in constructing the execution trace effectively identifies them. Measures of test effectiveness can be defined in terms of these executable elements. One such measure is the fraction of the elements exercised by a test case or set

of test cases. Another such measure is the fraction of the executable pairs of these elements exercised by a set of test cases. Since identifying the elements exercised also identifies the elements not exercised, it provides information for designing test cases to exercise the previously unexercised elements. Automated tools to support generation of test cases were investigated on RADC contract #F30602-76-C-0255.

Testing the SEMANOL specification of a programming language, guided by measurement of test effectiveness, can test the specification to a high degree of thoroughness. Testing, in which all executable elements and pairs of executable elements have been exercised at least once, will have tested the language specification to an exceptional degree of thoroughness. In such testing, many executable elements and executable pairs of elements will have been exercised more than once and many higher order sequences of executable elements will have been exercised. Such testing, guided by measurement of its effectiveness, can also eliminate redundant testing - i.e., test cases exercising the same elements - thereby holding down testing cost.

The test cases developed to test the SEMANOL specification also form a test for compiler conformance to the specification; i.e., the test cases may be compiled by the compiler under test and the execution results of the compiled test cases compared with the results of execution of the same test cases by the SEMANOL program and the SEMANOL Interpreter. These test cases, each of which is relatable (by the execution trace) to a specific portion of the SEMANOL specification and hence to specific language features, will provide a test of the compiler encompassing all language features. Compared to the test cases in current compiler validation systems, they should have the following advantages:

- Their execution results are defined by execution by the SEMANOL specification (semantic operator) and therefore are consistent with the language definition.
- They are more complete, since they meet an objective measure of effectiveness.
- They are less redundant and hence more cost-effective.

These test cases, although providing a thorough test of compiler conformance to the specification, do not necessarily provide a thorough test of compiler execution; i.e., they may not exercise all elements of compiler code. They may, however, be usable in other aspects of compiler testing - e.g., performance testing.

Relation of SEMANOL Specification to Axiomatic Specification

The axiomatic method¹⁰ of formal language specification was developed to provide a means for proving properties of programs and has been used to aid the formal verification of programs. It generally makes no attempt to represent syntax. It defines properties of the language in terms of axioms - statements in predicate calculus asserted to be true for programs written in the language. Usually, axioms are written defining properties of data types, functions, procedures, and statements. In the axiomatic specifications written to date - e.g., for PASCAL¹¹ and EUCLID¹² - implementation dependencies and execution effects are incompletely represented.

An axiomatic specification corresponds to a portion of the semantic definitions section of a SEMANOL specification. Although incomplete, axiomatic specifications are directly usable in current formal verification methodologies and hence are important to supporting such efforts. Preparation of axiomatic specifications by the DOD common language design contractors should identify ambiguities and semantically awkward constructions in the designs, contributing to the production of cleaner, more complete designs. Because an axiomatic specification emphasizes different aspects than a SEMANOL specification, the two specifications will complement each other, providing together the most complete, precise, and useful specification of any language to date.

A Standard Definition for the DOD Common Language

A standard definition of the DOD common language, meeting the requirements defined in the Environmental Requirements document, would comprise:

- a formal SEMANOL specification of the language,

- an English reference manual, prepared using the SEMANOL and axiomatic specification as a basis for writing accurate and complete descriptions of language features,
- an axiomatic specification suitable for use in formal verifications,
- a set of test cases, thoroughly exercising the SEMANOL specification, for validating compiler conformance to the specification.

The SEMANOL specification, being precise, complete, and executable, forms the base for the standard definition. The axiomatic specification, providing an alternate expression of a portion of the semantic definitions, should aid in the interpretation of the SEMANOL specification, as well as providing semantic definitions directly usable in formal verification methodologies. The English reference manual, prepared using the SEMANOL and axiomatic specifications, should provide an informal, but reasonably accurate and complete, description of the language features. The set of test cases, having been used to test the SEMANOL specification, can be used to confirm, by execution results, interpretations of language features; and the test cases are usable for validating compiler conformance to the specification.

Use of the Standard Definition to Support Language Control

A standard definition, as described in the preceding section, should contribute substantially to the effectiveness of the standardization effort. By providing a precise, complete, visible, and executable definition in the early stages of language development, it should provide a much more sound base for the DOD common language than has existed for any other programming language, eliminating ambiguities and interpretation problems that have plagued the early years of most programming languages.

The SEMANOL specification can be used as the specification for compiler implementation. A reference manual, prepared using the SEMANOL specification and consistent with it, should aid in interpreting the specification. Having this precise and complete specification (carefully prepared not to over-constrain the implementation) should aid compiler writers in implementing the language as intended. The test cases, providing a thorough test of

compiler conformance to the specification, should contribute to the early existence of compilers correctly and uniformly implementing the language.

Most programming languages have, in their early stages, numerous problems - problems in understanding how to use the language, problems in faulty implementations, problems in programming manuals inconsistent with implementations, and problems in achieving the language objectives. The standard definition should not only help in minimizing these problems but also contribute to their resolution. As problems are reported by early users, these problems can be analyzed using the standard definition. If the problem report contains a code example of the problem, that code can be executed using the SEMANOL specification and the SEMANOL Interpreter, with the execution trace identifying the portions of the language involved. This can help determine whether the problem is one of user interpretation, faulty implementation, or is a real language usage problem. If that last case holds, the specific language features contributing to the problem can be identified. Any proposed solutions involving language changes (which may be encountered despite the intent to have no changes) can be analyzed by writing the changes in SEMANOL, determining their effect on the SEMANOL specification (revealing, perhaps, unexpected side effects), and testing their effect by executing the proposed change. The result should be more definitive analysis and sounder decisions.

A substantial portion of the work on SEMANOL has been supported by RADC, including development of SEMANOL specifications of JOVIAL (J3), JOVIAL (J73), and BASIC, development of the SEMANOL Interpreter, and investigation of automated tools to support generation of compiler test cases. The author wishes to acknowledge the value of discussions with S. DiNitto and Capt. J. Ives of RADC and with E. Anderson, F. Belz, P. Berning, E. Blum, and D. Heimbigner of TRW.

References

1. Department of Defense Common Language Environmental Requirements, Preliminary Version Distributed to DOD Higher Order Language Working Group.
2. Revised "Ironman" Technical Requirements for DOD Higher Order Computer Programming Languages.
3. E. K. Blum, Towards a Theory of Semantics and Compilers for Programming Languages, J. Computer and System Sciences 3 (1969), 248-275.
E. K. Blum, The Semantics of Programming Languages, Part I, TRW-SS-69-01, December 1969, Part II, TRW-SS-70-02, December 1970.
E.R. Anderson, F.C. Belz, and E.K. Blum, SEMANOL(73), A Metalanguage for Programming the Semantics of Programming Languages, Acta Informatica 6, 109-131 (1976).
E.R. Anderson, F.C. Belz, and E.K. Blum, Issues in the Formal Specification of Programming Languages, IFIP WG2.2 Bulletin 1977.
4. E.R. Anderson, SEMANOL(76) Interpreter, RADC-TR-77-365, Vol. IV, Nov. 1977.
5. F.C. Belz and I.M. Green, SEMANOL(76) Specification of JOVIAL(J3), RADC-TR-77-365, Vol. III, Nov. 1977.
6. P.T. Berning, SEMANOL(73) Specification of JOVIAL (J73), RADC-TR-75-211, Vol. III.
7. SEMANOL(73) Specification of Universal CMS-2, USN Contract #N00123-74-C-1878, May 1975.
8. F.C. Belz, R.M. Hart, D.M. Heimbigner, Minimal BASIC SEMANOL(76) Specification Listing, RADC-TR-77-170, Vol. II, May 1977.
9. F.C. Belz, SEMANOL(76) Reference Manual, RADC-TR-77-365, Vol. II, Nov. 1977.
10. C.A.R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM 12, No. 10, Oct. 1969.
11. C.A.R. Hoare and N. Wirth, An Axiomatic Definition of the Programming Language PASCAL, Acta Informatica 2, 335-355 (1973).
12. R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J. Popek, Proof Rules for the Programming Language EUCLID, May 1977.

APPROACH TO IMPLEMENT A FAMILY OF
COMPILERS FOR A HIGH ORDER LANGUAGE

(Notes)

Hartmut G. Huber
15 June 1978

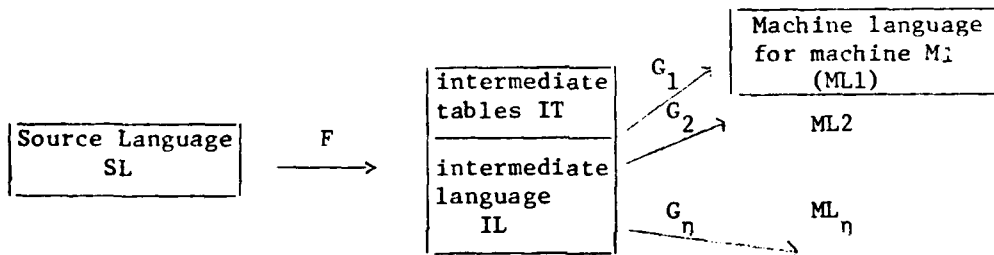
Approach to Implement a Family of Compilers for a HOL

The DOD HOL will be implemented on many machines, both military and commercial (say 10 - 20 computers). The task of building these compilers, certifying them and maintaining them is formidable if they are all independent. The design outlined in this draft is aimed to reduce cost and complexity of the following tasks:

- (1) building compilers
- (2) certifying compilers
- (3) maintaining compilers

A second goal is to increase the reliability of each compiler and to insure uniform performance of all compilers.

The approach is certainly feasible. It is a matter of judging how practical it is and of weighing advantages against disadvantages.



Each compiler C_i consists of the same frontend F and a code-generator G_i for a particular machine.

$$C_i = G_i \circ F$$

Also, a set of routines R_i required for runtime support is associated with each compiler C_i .

F and each G_i is written in SL; R_i will, in general, contain machine language routines besides routines written in SL.

Requirements for F

- All data structures use 32 bit words.
- Constants are stored as character strings.
- Modular design that allows forming a compiler with many overlays if necessary.
- Upper limit for size of parsing tables and other tables that must reside in memory as a whole.

Requirements for IL

ICF (= Intermediate Code File = storage space for IL) does not have to be resident in memory as a whole, it must allow "paging",

All control structures and substructures must be recognizable by special marker entries,

The IL Code is correct, that means, the code generator does not have to cope with bad input,

Entries representing references and operators have a reference count to be used for register and temporary management in a subsequent code generator,

Target machine independent optimization is already done,

IL code must be easily manipulatable, movable by an optimizer; this suggests that entries be quadruples; operator, two operands, result,

IL code is sequential, all nesting is resolved,

IL code does not contain forward references except for forward jumps.

Requirements for IT

The intermediate tables must contain the following information:

Symbol table

Constant table

Compiler generated label table,

Preset values associated with symbols that are to be initiated.

No memory allocation is done at this point.

Code Generators

Many code generators will have major parts in common, e.g. memory allocation and temporary management for machines that have the same word length. Other parts are unique for each code generator.

Thus the effort for constructing n code generators will not be n times the effort for one code generator. Ideally, one would have one general program that would construct a code generator for a machine, given a description of the relevant characteristics of M . We at NSWC are very interested in a solution to this problem.

Bootstrapping

Assumption: A first compiler exists, written in SL, running on machine Mo: $Co = G \circ F$.

Steps to construct compiler for machine Mi:

1. Write Ci for Mi in SL
2. Compile $G \circ F$ on Mo $\rightarrow C(Mo)i =$ compiler running on Mo, producing code for Mi
3. Compile $G \circ F$ on $C(Mo)i \rightarrow C(Mi)i = Ci$
= compiler running on Mi, producing code for Mi.
4. Write runtime system Ri for Mi.

Certification of a new compiler

Let $T1$ be a set of self testing test programs, $T2$ a set of programs for which code comparison is done. $T1$, $T2$ may overlap.

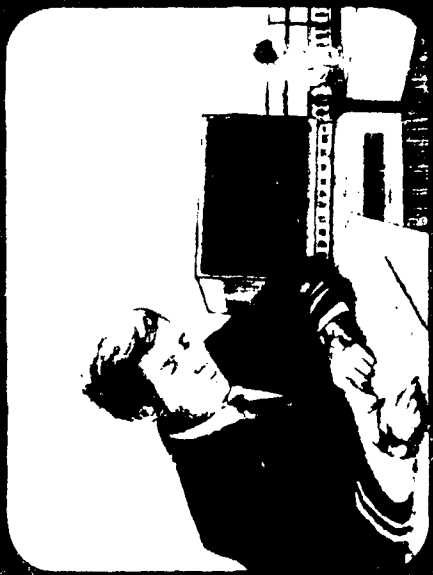
Certification steps:

1. Compare Code on the IL level for all programs in $T2$ using Co and Ci
2. Compare Code on the ML level for all programs in $T2$, using $C(Mo)i$ and Ci .
3. Execute self testing tests on Mi.

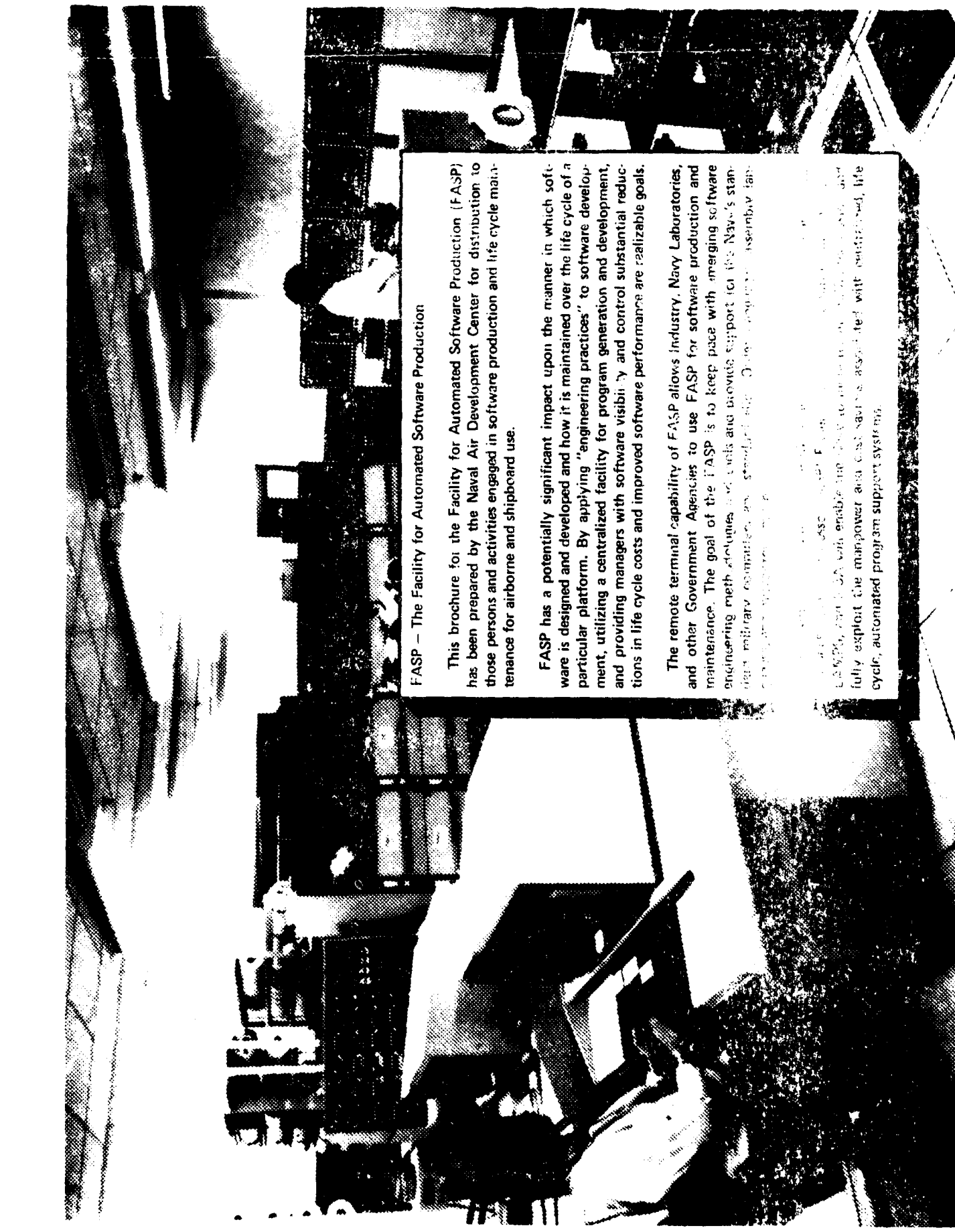
Starting up the Development of a Family of Compilers

1. Select a system programming language SP (e.g. AED) and write a base compiler in SP, possibly only for a subset of SL.
2. Rewrite the frontend of the compiler in its own language (use a subset if base compiler was implemented only for a subset). This produces F .
3. Select machine Mo and write code generator Go in SL.
4. Write runtime system Ro for machine Mo.

After these steps a STANDARD COMPILER is available on machine Mo to serve as a basis for the development of the family of compilers for a family of machines.



espa



FASP — The Facility for Automated Software Production

This brochure for the Facility for Automated Software Production (FASP) has been prepared by the Naval Air Development Center for distribution to those persons and activities engaged in software production and life cycle maintenance for airborne and shipboard use.

FASP has a potentially significant impact upon the manner in which software is designed and developed and how it is maintained over the life cycle of a particular platform. By applying "engineering practices" to software development, utilizing a centralized facility for program generation and development, and providing managers with software visibility and control substantial reductions in life cycle costs and improved software performance are realizable goals.

The remote terminal capability of FASP allows industry, Navy Laboratories, and other Government Agencies to use FASP for software production and maintenance. The goal of the FASP is to keep pace with emerging software engineering methods and tools and provide support for the Navy's standard military computer architecture. Other capabilities include assembly language programming, debugging, and testing.

FASP is a unique facility which enables the Navy to maintain a high level of software production and maintenance. It is a centralized facility which provides support for the Navy's standard military computer architecture. Other capabilities include assembly language programming, debugging, and testing.

Overview of FASP

FASP, the Facility for Automated Software Production, is a NAVAIRDEVCON facility in which operational and system test software for any Navy platform can be developed and maintained. This facility offers:

- **As Hardware** — The equipment of the NADC Central Computer System (CCS), a large complex of commercial computers, including two CDC 6600's and one CDC CYBER 175, plus extensive supporting peripheral devices.
- **As Software** — A full suite of program-generation capabilities for standard Navy languages and target machines, together with tools to support the use of modern software engineering technology.

As a software generation facility, FASP complements the capabilities of individual platform integration facilities where the operational hardware configuration is mocked-up in a simulated environment for testing, evaluation and training (figure 1).

Everybody Needs Software

Each Navy project requires the use of a software generation facility (SGF) to support software design, development, testing and maintenance. FASP provides a software generation environment suitable for all Navy project needs. FASP, unlike the typical SGF, offers additional services that support the management of programming activity, and assists in many individual programmer tasks which are part of software production. FASP can provide

these additional capabilities because the return on the development investment is realized over a large and growing user base comprising many Navy projects.

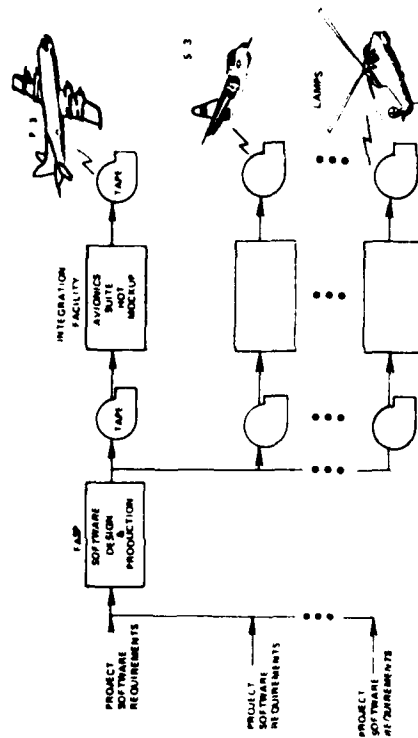


FIGURE 1. FASP SUPPORTS MULTIPROJECTS

The Motivation for FASP

The software problems of poor quality, late deliveries, and especially, increasing life-cycle costs provided the original motivation which led to FASP. The software generation function offered a natural leverage point whereby new software technology developments could be applied to benefit all problem areas. The approach was to:

- increase management visibility of software and provide a means for measurement and control;
- increase programmer productivity through automation;
- use new concepts of structure and modularity to improve quality and to reduce the cost-of-change during maintenance.

Historically, the typical software generation facility has been centered around the same military computer hardware which supported the operational needs of the project. The expensive military hardware was selected to meet the mission requirements with minimal consideration, if any, given to the production requirements of the operational software. Typically military computer hardware provided limited processing throughput, limited capabilities in peripheral devices, limited memory, and had very limited support software when viewed in terms of the needs of a large programming staff.

Speeds Development

FASP was designed to operate on a large-scale, commercial computer where, due to the large investment of industry in a competitive market, the limitations of the military computer have been eliminated (figure 2). In this way FASP frees the software development effort from total dependence upon the target hard-



FIGURE 2. LARGE-SCALE COMMERCIAL COMPUTERS

ware availability and reduces the number of conflicting demands for target hardware access.

The creators of large software systems, in particular the programmers, make decisions at a tremendous rate. Individually, the decisions are not difficult but each constitutes an opportunity for error; and because of their sheer number, some are made incorrectly. Those that are made incorrectly and escape detection in the debug and test phase later reveal themselves as software failures.

SOFTWARE ENGINEERING TECHNOLOGY BASE

- HIGH ORDER LANGUAGES
- TOP-DOWN APPROACH
- STRUCTURED PROGRAMMING
- SOFTWARE VALIDATION AND VERIFICATION
- PROGRAM EVOLUTION DYNAMICS
- COST AND PERFORMANCE DATA COLLECTION
- MANAGEMENT VISIBILITY AND CONTROL
- PROGRAMMING STANDARDS AND CONVENTIONS
- DATA BASE MANAGEMENT SYSTEMS
- METHODOLOGIES, TOOLS, TECHNIQUES AND PROCEDURES

There are software engineering methodologies for producing relatively error-free programs. These methodologies are widely acknowledged to be effective. Because of facility limitations, schedule constraints and short-range economic pressures, they are

less widely practiced. FASP provides a formalized structure within which there appear, naturally and conveniently, the software engineering tools that are often slighted under the pressures of time and money.

Cost Effective Approach

In improving the software quality, FASP simplifies the problems of software maintenance. Also, FASP insures continuity from development by the System Prime Contractor or Principal Development Activity through maintenance by the Navy; this continuity encompasses both facilities and support tools. The same tools and records that support the development remain available throughout the maintenance phase, thereby minimizing transition problems and training.

Under FASP the tools which support standard Navy computers and languages are available to all users, although the software for a project is fully protected from unauthorized access. A new project can thus obtain a software generation capability at a minimum cost and apply its full resources to developing the operational software. The power of the NADC CCS is sufficient to handle peak loads and this power also provides a "safety-valve" to handle unplanned development tasks. Furthermore, new projects using FASP will add additional capabilities to meet new requirements, thus contributing to the continuing development of FASP while taking full advantage of capabilities already in place. In this way FASP offers a continually growing range of services appropriate to all projects. The FASP concept of centralized, multi-project support is the most cost-effective approach to life cycle software support for the Navy.

The Facility Aspects of FASP

FASP utilizes the large-scale, third-generation, commercial computers of the NADC Central Computer System (CCS). This is a multiframe configuration offering multiprocessing and CYBER 170 Model 175, operate independently, but are able to access 500,000 words of shared memory (Extended Core Storage) and share all CCS peripheral devices.

The CCS suite of peripheral devices currently includes:

- 784 million words of mass storage through 4 drums and 52 disc drives
- 96 I/O ports to service remote batch and interactive terminals
- 12 tape drives
- 3 high-speed card readers and 2 card punches
- 5 high-speed line printers
- 14 displays
- special real-time interfaces.

A total of 40 Input/Output Computers (Peripheral Processors) interface between the Central Processors and the peripheral equipment to handle all service demands. Each I/O computer is connected to all of the peripheral equipment over a shared peripheral linkage, so that any processor can communicate with any peripheral device; device selection takes place automatically by electronic switching.

Easily Accessible

Located at NADC, the CCS supports an extensive terminal network at Government and industry sites across the nation (figure 3). Each remote batch terminal, interactive terminal, or intercomputer link allows direct access to the CCS equipment and

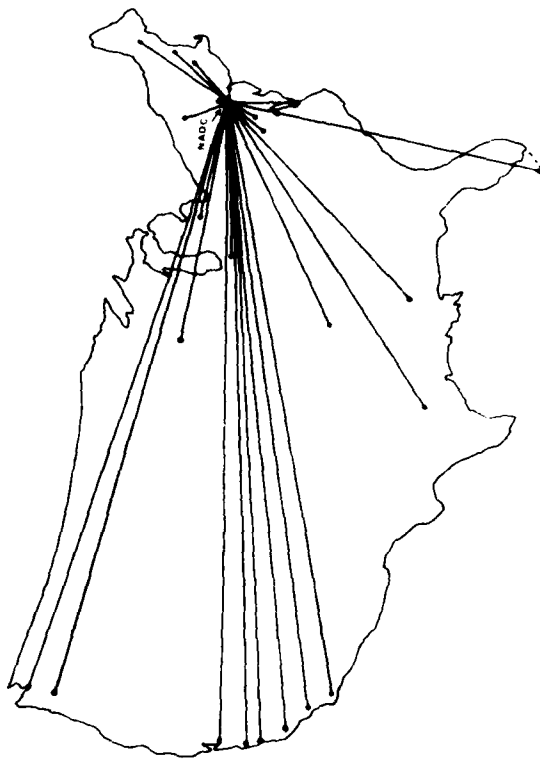


FIGURE 3. FASP REMOTE TERMINAL SITES

FASP capabilities from the remote site. A project's work can be accomplished on-site, at other Navy laboratories, at contractor's sites or at some combination of such sites. Compatible terminals are commercially available or can be provided to a contractor as Government Furnished Equipment, whichever is more cost-effective. Thus, the Principal Development Activity (PDA), System Prime Contractor (SPC) and the Support Software Activity (SSA) can each use FASP through a local extension of the NADC CCS (figure 4).

The NADC CCS is continually upgraded to keep pace with the resource requirements of FASP users. Established priority and control procedures assure responsive computer service to the user community over a multi-shift operation. Projects using the CCS computer services incur costs which are proportional to their actual resource usage rather than at a fixed, flat rate. In contrast, a project using a dedicated software generation facility must pay the full cost of that facility, whether the usage is minimal or saturates the system. With a dedicated facility, around-the-clock access to the computer is gained at a premium price; the CCS multishift access is available at no additional cost to the user, with better turnaround time during nonprime hours.

Supports Project Integration Facilities

An important product of FASP is the load tapes generated for use on the target hardware. Although many projects share the facilities of the NADC CCS through FASP, each project continues to operate a dedicated integration facility where a hot mockup of the actual avionics or shipboard equipment is supported with realistic simulation of external inputs. The integration facilities are distinct for each project because the actual equipment is vastly different for each platform. Typically, the integration facilities serve as the hardware configuration baseline and are used for laboratory integration of the software and hardware, evaluation of man/machine functions, and test and evaluation of Engineering Change Proposals. The simulation of realistic external inputs allows the total system to be tested in a laboratory environment where sophisticated instrumentation equipment is accessible. This approach minimizes costly flight or shipboard testing where integration and checkout take place under much more difficult circumstances.

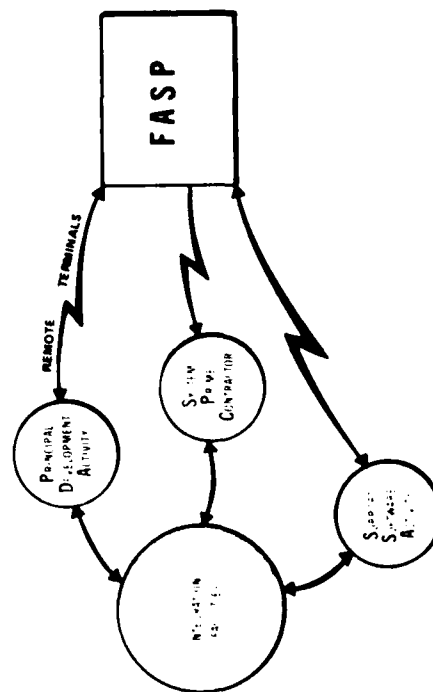


FIGURE 4. FACILITY/ACTIVITY INTERACTION

The Software Aspects of FASP

The FASP is a comprehensive software generation facility consisting of an integrated collection of software development and maintenance tools (figure 5). The FASP tools are designed to provide support for each phase of the software life cycle. The five generic types of FASP tools are (1) design, requirements and specification aids, (2) implementation tools (translators and system generators), (3) testing tools, (4) project management tools and (5) configuration management tools.

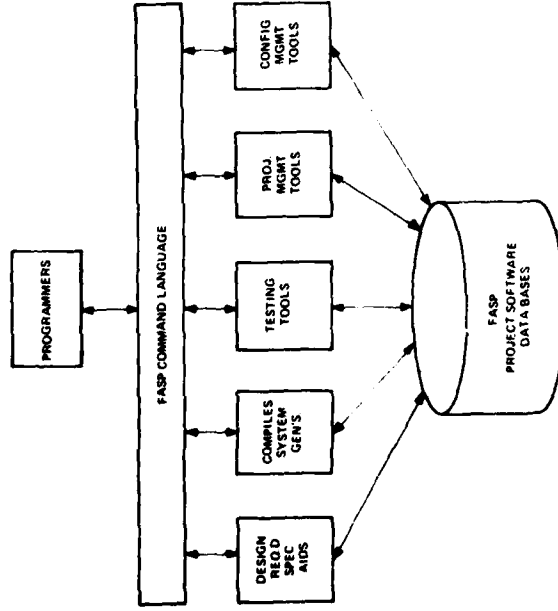
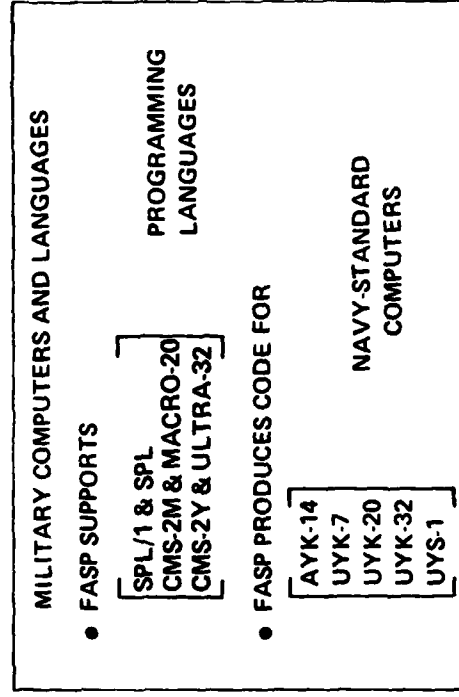


FIGURE 5. FASP TOOLS

The FASP implementation tools support the Navy's standard programming languages and military computers. Thus the FASP system responds to many system design needs.



Under FASP, the mission software being developed resides in a project data base consisting of a number of libraries with a master directory. In addition to being a data bank for the usual program data (source and object libraries), the FASP data base libraries contain program module interface data, test inputs and test results, modification histories and extensive software production data. The data base provides complete information on the operational program, how it developed, how a test or operational version is built, and what data is used to test it. In traditional software generation systems much of this information is kept informally by programmers, if it is kept at all.

FASP provides a uniform interface through a command language to all the necessary software development and maintenance tools that are part of the FASP System Software.

The FASP Processor handles the details of interfacing these tools with such additional tools as text-editors, simulators, test text-editor, compiler, system generator and simulator are managed smoothly by FASP so that communication from one tool to the next requires a minimum of programmer-specified data.

Encourages Modular Design

A project has complete freedom to set its own naming and coding conventions independently of other projects using FASP; in fact, FASP helps to insure that such conventions are followed, because it allows greater management visibility into the software. Together, the FASP commands and the data base structure actually encourage the use of modular design techniques and shared common libraries through the use of module-oriented organization in both source and object libraries and through the use of object library managers. Similarly, FASP supports the use of other modern software engineering practices such as partial recompilation in lieu of patching.

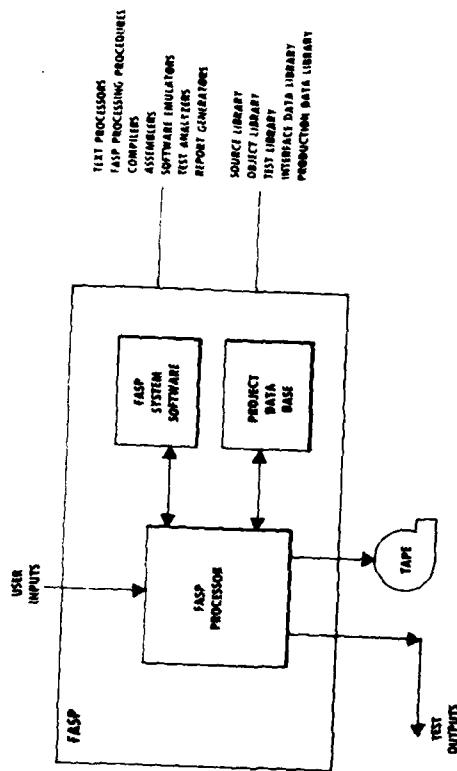


FIGURE 6. FASP PROCESSING

The uniformity from one project data base to the next guarantees that all mission software components are treated in a like manner, with the same kinds of support and descriptive data maintained for each component. This insures that system integration can proceed smoothly because all programmers have adhered to the same conventions and formats for their software. It also provides continuity so that one programmer can more easily take over responsibility for software developed by another programmer, or so that testing and maintenance activities can continue with maximum support and minimum interruption once software is delivered from a development group to a maintenance group.

The Automation Aspects of FASP

Software production is a labor-intensive field where the general principles of automation can be applied to reduce costs. In the FASP environment the programmer only has to think in terms of what needs to be done, rather than on how it can be done. Each of the FASP user commands automatically performs a particular programmer task, typically a multi-step sequence of operations. Throughout this sequence of operations, FASP automatically performs appropriate record-keeping by storing in the data base descriptive information that is a natural byproduct of the software generation.

The data base, then, is not simply a repository for the actual project software under development, but also a source of technical support data and management information which reveal the genesis and current status of the project software. The continuity of format and contents over all data bases provides to management a visibility not otherwise possible, while the uniformity of operating procedures (FASP commands) provides to management an assurance against deviant programmer actions or catastrophic losses of either the software itself or that data without which the software cannot be utilized.

To the programmer FASP acts both as a global executive through which all these software engineering tools can be invoked and as a global data base manager through which the mission software data bases are maintained, updated and protected (figure 7). FASP is a complete program generation subsystem consisting of a full spectrum of job profiles, requiring minimal knowledge of or training in the full complexity of KRONOS commands. FASP jobs run, as do all other CCS jobs, under the standard KRONOS operating system.

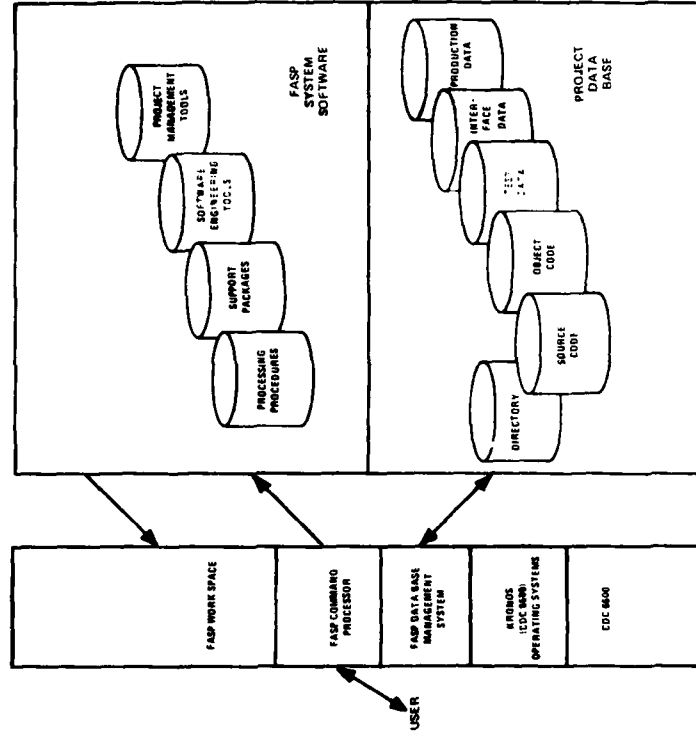


FIGURE 7. AUTOMATION OF FASP OPERATIONS

FASP supports the concept that system generation directives, test data, and other essential program data should be saved and protected for later re-use. In order to promote this, support data is automatically collected and maintained in the FASP data base with minimal programmer intervention. Some components of the data base are completely specified by the programmer; the source library which contains the programs designed and coded by the programmer is in this category. However, other components are supplied and updated by FASP itself as a service to programmers (and their managers); the modification history which details the growth and change of the software under development or maintenance is an example of this category. The data base is a source of technical support data and management information which reveal the history and current status of the project software. This information provides management with the means for production control.

FASP provides a simple user interface such that many strictly clerical duties can be assigned to a program librarian (figure 8). This frees the professional programmers from filling out coding forms, keypunching, job submission, retrieving computer output and allows them to fully concentrate on producing programs. The use of program librarians reduces the total labor costs on most large-scale software projects.

FASP extends the same automated support to both classified and unclassified jobs. Offsite classified processing is possible through the use of secure terminal lines from remote sites with NAVELEXACT clearance and the appropriate cryptographic equipment. Both confidential and secret processing can be accomplished through FASP.

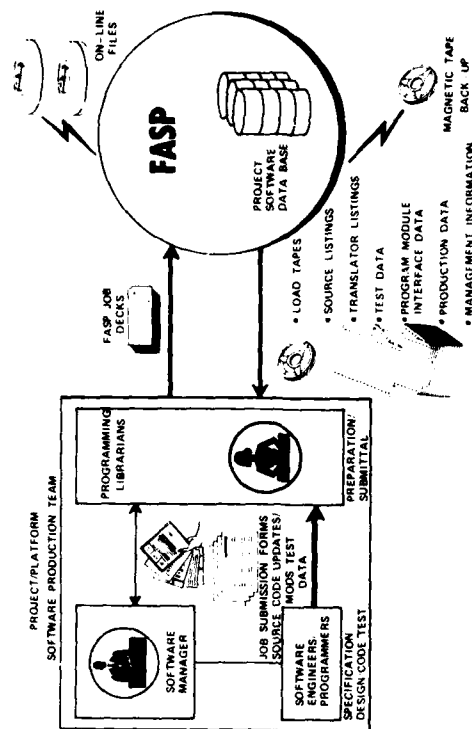


FIGURE 8. AUTOMATED SOFTWARE PRODUCTION

The Production Aspects of FASP

On a detailed level the production of the mission software involves successive steps of translation, system generation and testing. During translation the compiler (or assembler) which takes the source form of the code and produces the object form is actually a cross-compiler (or cross-assembler) executing on one machine (the 6600's of the NADC CCS) but producing code for another machine (the particular target hardware such as an AYK-14). In the FASP environment there are two kinds of testing: simulation testing within FASP and actual testing on the target computer in the integration facility (figure 9).

Early Detection of Software Errors

Historically, all software testing was accomplished in the integration facility in a hands-on mode on the military computer

hardware. FASP allows for unit testing of modules on the CCS by providing simulations of the standard military computers. This allows the programmers to eliminate many software errors early in the development cycle. Simulation testing makes it possible for many programmers to proceed with testing simultaneously without requiring sequential, exclusive access to the target hardware. Because FASP is a multiprogramming system each programmer has a pseudomachine available for his exclusive use on demand, thus allowing concurrent test runs by many programmers.

The high throughput and remote terminal capability of the CCS gives the users of FASP quick turnaround time, a critical factor in meeting project deadlines and improving programmer productivity. The remote terminals are located close to programming staff resulting in immediate access to FASP for computer input and receiving printer output.

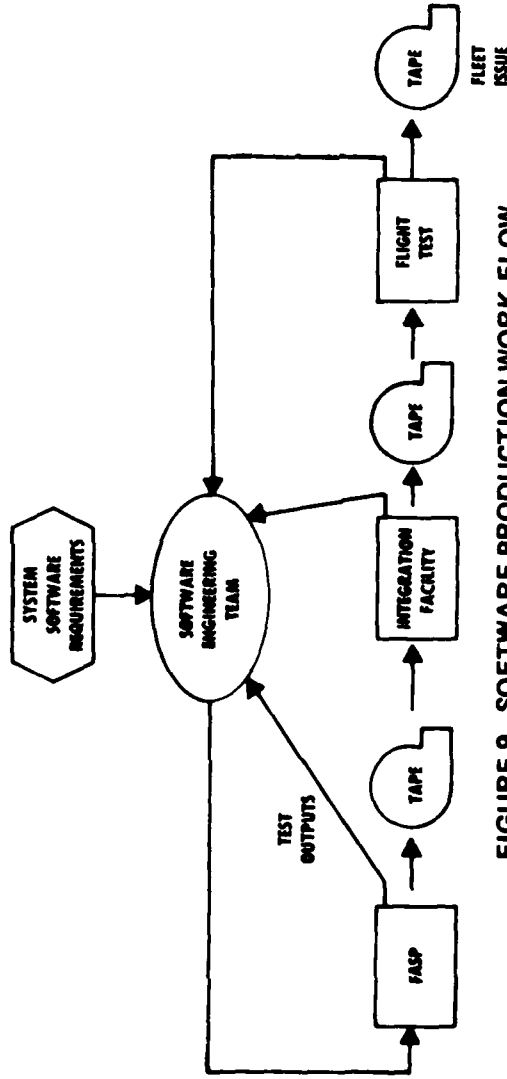


FIGURE 9. SOFTWARE PRODUCTION WORK FLOW

Throughout these steps in the production of software, project managers are interested in planning, monitoring, and controlling the work of their programmers. A manager's ability to control production is dependent on his ability to collect data on the status of the software and the resources expended to achieve that status. The FASP Project Management Tools provide for both the collection and the reporting of this data (figure 10). Through production data stored in the FASP data base, the manager has a means to define and control the software configuration, assess incremental progress, detect trends and anomalies, track changes, and ensure the observance of programming standards and conventions. This data can be used for baselining, accounting, and design change control throughout the project.

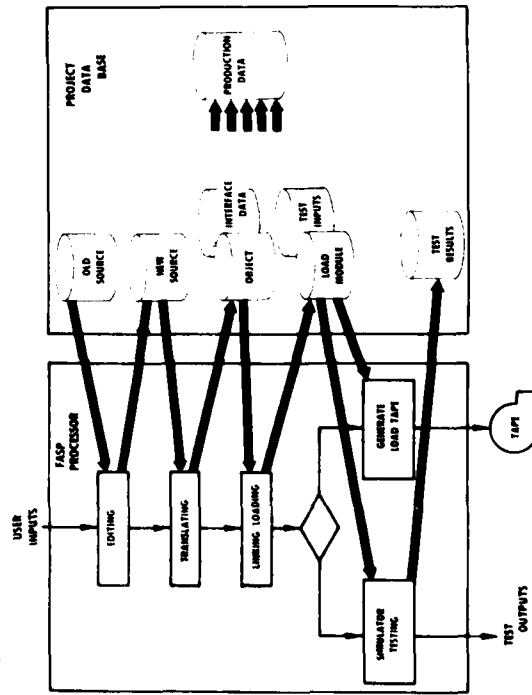


FIGURE 10. COLLECTION OF PRODUCTION DATA

Control Plus Visibility

The standard report generator supported by FASP produces a series of Software Management Reports that provide financial information, comprehensive utilization data and extensive software status data (figure 11). The software status reports are generated for all or a selected number of project software data bases. The reports generated by FASP provide the visibility into software production which managers often lack.

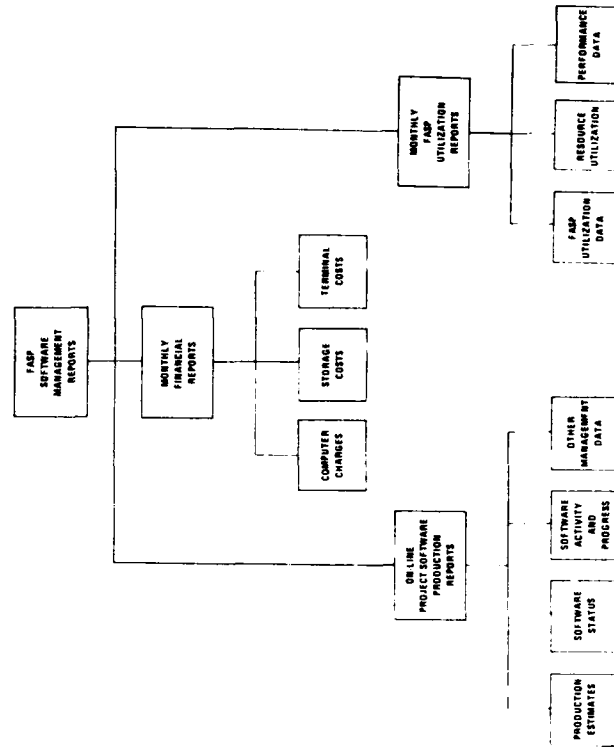


FIGURE 11. FASP REPORTS

Life Cycle Support

The activities of translation, system generation, and testing, along with parallel management activities, continue throughout the total software life-cycle which divides into a development phase and a maintenance phase (figure 12). Development covers design, implementation, and checkout. Maintenance, which in the hardware world means prevention and detection of component failure caused by aging and physical stress, means modification to meet changing requirements or to correct newly detected errors, in the software world.

Reduced Maintenance Costs

Recent military software projects have shown that typically three-quarters of the total life cycle costs go toward maintenance. These costs are directly related to the maintainability characteristics built into the mission software during development and to the kind of software generation facility used for the maintenance work. FASP provides a development environment favorable to the application of those software engineering techniques which contribute to the maintainability of software.

While some factors may differ in relative importance between development and maintenance, basically the same software generation facility requirements are dictated by both phases of the software life-cycle. FASP provides facility, support tool, and data base continuity from development to maintenance, so that even if the responsible agency/contractor changes, the mission software support environment remains constant. Moving mission software from one software generation facility to another is costly, parti-

cularly if it entails redesigning and reconstructing a software generation facility. The continuity of support tools implies that all translation, system generation, and testing activities are as fully supported for maintenance as they are for development. Data base continuity provides to the maintainers complete libraries of mission software and support data which were necessary for development so that the maintainers do not suffer from a lack of essential or contributory data.

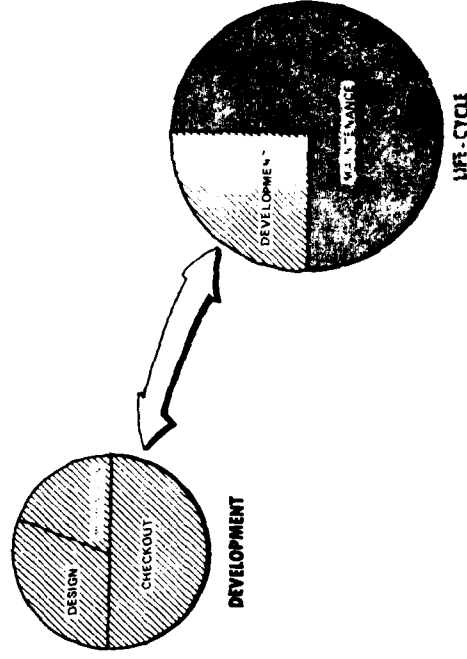


FIGURE 12. SOFTWARE LIFE CYCLE COST DISTRIBUTION

Further Information on FASP

Personnel interested in obtaining FASP documentation represent a wide spectrum of management responsibilities and software backgrounds ranging from corporate level managers and program managers to software engineers and programmers. A hierarchy of FASP documents has been defined to match individual requirements. The documents include this brochure as well as a FASP Management Summary, FASP Software Production and Maintenance Methodology, FASP Programmer's Handbook, FASP User Manuals and other documents containing detailed user information (figure 13).

The staff of the Advanced Software Technology Division of NADC stands ready to provide assistance to any project in evaluating FASP support for that project, determining how special project needs can be accommodated, or answering questions relative to FASP capabilities. Arrangements can be made for projects wishing to explore or evaluate the use of FASP. Training courses are available and arrangements can be made to conduct the training at user sites.

For further information contact:

**Mr. Henry G. Stuebing, Code 503
Advanced Software Technology Division
Software and Computer Directorate
NAVAIRDEVCEEN, Warminster, Pa. 18974
Phone (215) 441-2314 or AUTOVON 441-2462**

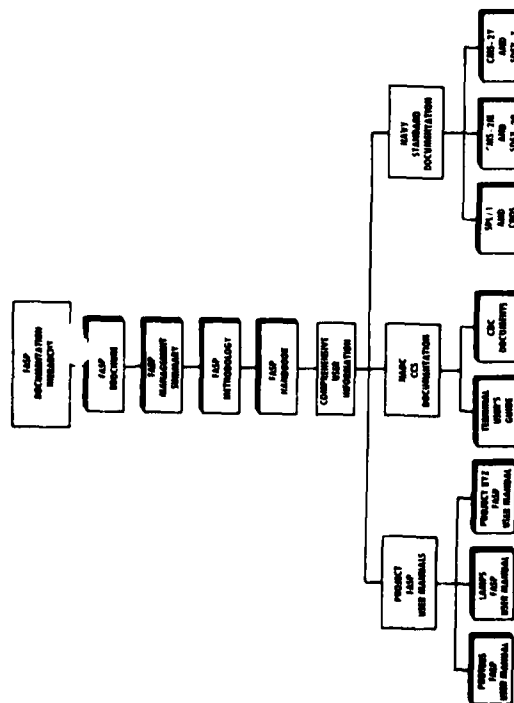


FIGURE 13. DOCUMENTATION HIERARCHY





DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

1400 WILSON BOULEVARD
ARLINGTON, VIRGINIA 22209



TO THE PARTICIPANTS IRVINE WORKSHOP

You have recently received notification from Dr. Standish of the University of California-Irvine of a meeting 19-22 June 1978. This workshop will explore technology appropriate for software development and maintenance environments.

The Department of Defense High Order Language Working Group (HOLWG) is engaged in the second phase of a design effort for a common language for DoD embedded computer systems. In expectation of the culmination of this effort next year, other portions of the program are being planned. As a part of this exercise, in order to initiate evolving thinking on a variety of topics an Environment Requirements document is being created. I am sending you a preliminary version of this document for your information and to suggest some topics for discussion at the workshop. Suggested modification and expansion of the document resulting from this meeting and other input will be entered into the text editing system, an updated version produced, and distributed.

We regard the environment, compilers, control, tools, etc. as key to the success of the entire common language program and, indeed, the focus of the entire DoD technology program to improve productivity in software. Your contributions at this early phase of the effort will be instrumental in determining the directions of research and technology during this vital period.

Sincerely,

RECEIVED

MAY 31 1978

T. A. STANDISH

William A. Whitaker

William A. Whitaker
LtCol, USAF
Chairman, High Order
Language Working Group

Department of Defense
Common Language
Environment Requirements

Prepared for DoD Higher Order
Language Working Group (HOLWG)
15 May 1978

TABLE OF CONTENTS

Section	Title	Page
1.0	Introduction	1-1
1.1	Purpose	1-1
1.2	Reference Documents	1-2
1.3	Definition of Requirements Terms	1-5
2.0	Language Standard	2-1
2.1	Standard Document	2-1
2.2	Intent of the Common Language	2-1
2.3	Explicit Policy and Controls for Standardization	2-2
2.4	Approach	2-3
3.0	Control and Support Organizations	3-1
3.1	Configuration Control Board	3-1
3.2	Compiler Validation Agency	3-2
3.3	High Order Language Working Group	3-3
3.4	Language Support Agency	3-3
3.5	Application Library Agency	3-4
3.6	User Organizations	3-4
4.0	Configuration Management	4-1
4.1	Objectives and Strategy	4-1
4.2	Configuration Control for the Common Language	4-1
4.3	Configuration Control for Translators	4-2
4.4	Configuration Control for Supporting Software	4-2
4.5	Configuration Control for Application Programs	4-3
5.0	Compilers	5-1
5.1	Production Compilers	5-1
5.2	Compilers Validation	5-2
5.3	Delivery Packages	5-3
5.3.1	Compiler Delivery	5-3
5.3.2	Compiler Delivery to User Site	5-4
5.4	Compiler/User Interface	5-4
5.4.1	Compiler Inputs	5-4
5.4.2	Compiler Outputs	5-6
5.4.3	Run Time Outputs	5-9
5.5	Translator Functional Objectives	5-10
5.6	Translator Production Guidelines	5-11
6.0	Run Time Supporting Software	6-1
6.1	Purpose and Intent	6-1
6.2	Executives	6-1
6.3	Test and Debug Package	6-3

TABLE OF CONTENTS (CONTINUED)

Section	Title	Page
7.0	Other Supporting Software	7-1
7.1	Purpose and Intent	7-1
7.2	Compile Time Tools	7-1
7.2.1	Fault Detection Tools	7-1
7.2.2	Flow Charters	7-1
7.2.3	Correctness Provers	7-2
7.2.4	Symbolic Program Executors	7-2
7.3	Object Program Link/Load Tools	7-2
7.4	Requirements Generation Tools	7-4
7.5	Design Tools	7-4
7.6	Construction Tools	7-5
7.6.1	Design Language Translation Tools	7-5
7.6.2	External Library Systems	7-5
7.6.3	Test and Debug Systems	7-5
7.7	Integration Tools	7-6
7.8	Control Tools	7-6
7.8.1	Configuration Management Aids	7-6
7.8.2	Project Control Tools	7-6
7.9	Migration Tools	7-7
8.0	Application Software	8-1
9.0	Language and Environment Documentation	9-1
9.1	Language Documentation	9-1
9.2	Compiler Documentation	9-1
9.3	Supporting/Application Software Documentation	9-2
9.4	Methods of Documentation	9-2
10.0	Information Collecting, Dissemination and Promotion	10-1
11.0	Training Support	11-1
11.1	Types of Training Required	11-1
11.1.1	Programmers Using the Common Language	11-1
11.1.2	Translator Developers	11-2
11.1.3	Management of Projects Using the Common Language	11-2
11.2	Training Modes	11-2
12.0	Project Management Aids	12-1
12.1	Requirements Definition and Tracking Aids	12-1
12.2	Design Aids	12-2
12.3	Construction Aids	12-2
12.4	Integration Aids	12-3
12.5	Control Aids	12-4

Section 1

Introduction

1.1 Purpose

The Department of Defense (DoD) has defined a Common Higher Order Language (HOL) for embedded systems based upon a language requirements document. The language requirements document is the product of the HOL Working Group (HOLWG) formed within DoD. It has incorporated comments and suggestions from the government, academic institutions, and industry until it was judged to be of sufficient correctness and thoroughness to be used as the requirements document for the design of a DoD common language for embedded systems.

In order for the common language to be successful in achieving the desired objectives, the environment in which it is used has to be conducive to its support. The environment includes all supporting activities for using any language to develop programs for all systems - small, medium and large. These aids include for instance:

1. Organizations and methods to control the language and promote development of tools
2. Compilers for converting HOL into machine language of the target computer
3. Tools to aid in the design, test and debug of application programs

4. Organizations and methods to research the use of the language and prepare for its next version
5. Materials and techniques for training users of the language
6. Methods for collecting, cataloging and disseminating information about the language and programs written in the language
7. Project management aids to achieve successful implementation of systems where success is measured over the life cycle

This document describes the requirements for the environment necessary to the success of the common language. It will go through a number of iterations, as the language requirements have, considering suggestions from all parts of the software world. It will also spin off more detailed requirements in specific areas such as tools or control.

The theme behind the inclusion of any topic has been to list all methods which have come to be recognized as necessary for the production of reliable embedded systems.

This is a preliminary document for circulation to generate comments and wide latitude has been allowed. Later versions will strive for greater rigor.

1.2 Reference Documents

- o Standard Definition Document for the Common Language (to be defined).
- o DoD Requirement for High Order Computer Programming Languages, Ironman, Revised July 1977.

- o DoD Requirement for High Order Computer Programming Languages, Tinman, June 1976.
- o DoD High Order Language Program Management Plan, January 14, 1977.
- o The Navy Fortran Validation System, Patrick M. Hoyt, AFIPS Volume 46, 1977.
- o Test Planning, R. Dean Hartweck, AFIPS Volume 46, 1977.
- o A Time for Cross-Compilers, Marcus L. Byruck, Datamation, May 1977.
- o High Order Language Standardization for Avionics, W. L. Trainor and H. M. Grove, I.E.E.E. NAECON, 1977.
- o Language Control Facility (LCF) Study, RADC-TR-76-386, Volume I: Component Requirements for the LCF, Volume II: Evaluation of the Software Tools for the LCF.
- o Managing the Development of Reliable Software, R. D. Williams, Proceedings International Conference on Reliable Software, April 1975.
- o Integration Engineering: An Approach to Rapid System Deployment, Robert C. McHenry and Jerry A. Rand, Technical Report FSD 77-0179, IBM Corporation, Gaithersburg, MD.
- o A Program for Software Quality Control, Paul Oliver, Proceedings AFIPS 1974, Volume 43.

- o Experience in COBOL Compiler Validation, George N. Baird and Margaret M. Cook, Proceedings AFIPS 1974, Volume 43.
- o PSL/PSA: A Computer - Aided Technique for Structured Documentation and Analysis of Information Processing Systems, Daniel Teichroew and Ernest A. Hershey, III, IEEE Transactions on Software Engineering, Volume SE-3, No. 1, January 1977.
- o Design and Implementation of Programming Languages, DoD Sponsored Workshop, Ithaca 1976, Lecture notes in Computer Science Number 54, Springer - Verlag.
- o DoD's Common Programming Language Effort, David A. Fisher, Computer, March 1978.
- o Software Tools: A Building Block Approach, NBS Special Publication 500-14, I. T. Hardy, B. Leong - Hong, D. W. Fife.

1.3 Definition of Requirements Terms

The following terms have been used throughout the text to indicate where and to what degree individual requirements apply.

1. Shall - indicates a requirement
2. Will - indicates a consequence that is expected to follow or indicates an intention of DoD
3. Should - indicates a desired goal but one for which there is no objective test
4. May - indicates a requirement to provide an option to the user
5. Must - indicates a requirement placed on the user by the language and its translators

Section 2

Language Standard

2.1 Standard Document

The second document referenced in Section 1.2 states the technical requirements for the common language. It is a set of functional requirements for a language appropriate to embedded computer applications (i.e., command and control, communications, avionics, shipboard, test equipment, software development and maintenance, and support applications).

The syntax and semantics of the language are described in a document which shall become the standard for deciding whether or not all compilers conform to the language specification. That document shall be referred to as the standard definition document. The Configuration Control Board shall maintain and interpret the document.

2.2 Intent of the Common Language

The goal of the common language is to reduce total costs of software incurred by DoD. This goal will be promoted by the following general design criteria for the language.

1. Generality - The language should be of a general nature applicable to a wide range of embedded systems computer applications.
2. Reliability - The language should promote, encourage, and enforce the use of techniques which lead to reliable software.

3. Maintainability - The language should emphasize readability and understandability of programs and lead to less costly maintenance.
4. Efficiency - The language should allow compilers which produce efficient object programs.
5. Simplicity - The language should reduce unnecessary complexity by means of uniform syntactic conventions and consistent semantic structure.
6. Implementation - The language should facilitate production of compilers that are easy to implement and are efficient.
7. Machine Independence - The language should strive for machine independence to make possible the portability of application programs.
8. Formal Definition - There should be a formal definition of the language for unambiguous control.

2.3 Explicit Policy and Controls for Standardization

Once the common language is defined it shall be added to the list of approved higher order languages in DoD 5000.31.

In order for the HOL to achieve expected benefits, there shall be no variants of the language. Organizations supporting the common language shall monitor and oppose any attempts at non-conformance to the published standard.

Registration of the language as a Federal Information Processing Standard will be done so that it may be used throughout all government organizations.

The supporting organizations will monitor worldwide activities in language development and participate to promote further standardization of the language. Submissions to the American National Standards Institute and the International Standards Organization is appropriate to expand the user base and further reduce the likelihood of variants.

2.4 Approach

All environmental elements will support the above goals for the common language. Elements necessary for success are described in subsequent sections as follows.

1. The primary necessity is an organization to control the language and promote development of its supporting software. Section 3 describes this organizational structure.
2. Methods for controlling the common language and its compilers are required to allow managed change when necessary for technical growth. These methods are described in Section 4.
3. Compilers to convert the language from HOL to target machine code are required. Requirements for various types of compilers are given in Section 5.
4. Tools required for application program development, control, and execution are described in Sections 6 and 7. Tools for design, debug, test, modifications of code, and execution control are necessary parts of a successful software environment.
5. Section 8 discusses requirements for application programs written in the common language. The methods

AD-A089 090

CALIFORNIA UNIV IRVINE DEPT OF INFORMATION AND COMP--ETC F/G 9/2
PROCEEDINGS OF THE IRVINE WORKSHOP ON ALTERNATIVES FOR THE ENVI--ETC(U)
1978 T A STANDISH DAA629-78-M-0219
UCI-ICS-78-83 NL

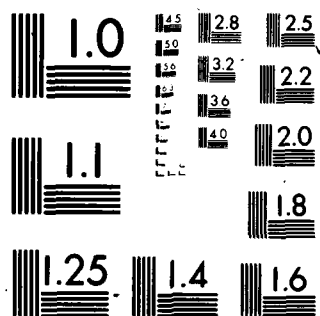
UNCLASSIFIED

4 of 4

AD A

10-80

END
DATE
FILMED
10-80
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

described will lead to high portability for embedded systems.

6. Documentation requirements for the common language and its supporting software are described in Section 9.
7. Section 10 describes methods for collection and dissemination of common language materials so that the embedded computer software community has ready access to all required common language information.
8. Diverse training will be required for successful implementation of the common language. Section 11 addresses these training requirements.
9. Section 12 describes the tools and techniques required by project management for successful embedded systems.

Section 3

Control and Support Organizations

The following paragraphs describe the organization of DoD agencies and user groups which have been proposed to effectively control standardization of the DoD HOL and provide support.

3.1 Configuration Control Board

A Configuration Control Board (CCB) shall be established by DoD and be responsible for custody and maintenance of the formal definition of DoD HOL. Primarily, the function of the CCB shall be to minimize changes to the language and prevent variant translators from occurring.

The CCB shall be the final arbiter in any interpretation dispute of the DoD HOL definition. All official interpretations shall become part of the language specifications. All requests for changes and interpretations will receive a prompt response.

To reduce potential influence of special interest groups, the CCB shall be autonomous to compiler or applications developers.

Membership of the CCB shall include representation from major Federal user communities within the United States. Membership from outside the U.S. will be appropriate as other nations make a major commitment to the language.

The CCB shall be operational as soon as the language is frozen and submitted for standardization. At that time, formal definition of the DoD HOL shall be controlled by the CCB.

3.2 Compiler Validation Agency

An agency will be established to validate that compilers are complete and correct implementations of the DoD HOL. The agency will not perform any function other than compiler validation and shall be independent reporting only to the CCB.

As a minimum, validation shall be conducted by subjecting compilers to a set of test programs. Development and maintenance of test programs shall be the responsibility of the validation agency. Trouble reports from users will be used to refine and update test programs in an effort to develop the most comprehensive test programs possible. All test programs shall be documented and made available to implementors who wish to test compilers independently prior to formal validation. Formal validation shall consist of reviewing compiler documentation and running the test programs.

A validation report shall be prepared by the agency.

Validation shall be required by defense projects when compilers are initially developed and when modified. A requirement shall be to validate the compilers at the start of any project which plans to use the compiler.

The validation agency shall be established and operational within one year from the selection of the single language.

3.3 High Order Language Working Group

A High Order Language Working Group shall have responsibility for coordinating DoD common language activities. This group will provide the services of initiating and coordinating research and development activities, incorporating feedback from language implementors and users into ongoing programs, and directing funding into the most productive areas. The group will provide overall guidance and direction to the Language Support Agency (LSA) and the Application Library Agency. The group will act as the liaison agent between the LSA and the CCB.

An ancilliary function of the group will be to represent the common language with national and international standardization organizations.

A member of the working group will chair an organization of members from government, academic institutions, and industry to research language problems, technical advances, and documentation methods.

3.4 Language Support Agency

A Language Support Agency shall represent the bulk of staff and funding in the DoD HOL organization. It shall be the focal point for most translator, support tool, and general library development and maintenance activity.

As the focal point for compiler, support tool and library development and maintenance, the LSA will be the primary interface for user and implementor communities. The agency shall develop and maintain documentation, develop and conduct training courses and respond to all user and implementor inquiries.

All compilers, support tools, libraries, and language documentation maintained by the LSA shall be readily available to any legitimate and qualified language user or implementor. Nothing will be done to inhibit language users and implementors from using the facilities and services of the LSA.

3.5 Application Library Agency

Application libraries, in the long term, will become very large and diverse. An effort will be undertaken to demonstrate the utility of a centrally supported application library. Several promising specialized application areas are signal processing, display processing, and communication networks. The ARPANET is a particularly well documented application that may be provided as a common application package.

Once developed, application libraries could be supported and maintained by their specific support agency. These agencies would be formed as desired for particular applications, in some cases colocated with the Language Support Agency.

3.6 User Organizations

User organizations are necessary to serve as a technical forum for common interests of those using the common language. All common language organizations will foster user organizations by giving them recognition, disseminating information about their meetings and purposes, participating in their meetings, and giving due consideration to all user organization proposals.

The organizations may be grouped by special interest such as the following.

1. Use of a particular translator

2. Use of a particular computer

3. A particular application area.

The user organizations will interface with the CCB for language change requests and with the LSA for technical information exchange.

Section 4

Configuration Management

4.1 Objectives and Strategy

The strategy for controlling the common language and environment will be to institute managed change which allows for evolutionary, stepped growth of the language, translators, development tools, and test tools. Application programs will continue to be controlled by the responsible agencies.

4.2 Configuration Control for the Common Language

Responsibility for all changes to the common language shall be under control of the Configuration Control Board (CCB).

All proposals for changes will be accepted and recorded, but, in general, changes will be discouraged due to the cost impact. If the change passes a threshold to determine that it will be considered by the CCB, it will be investigated for being part of a generic requirement. The CCB will be supported by the LSA to investigate the impact and necessity for any proposed changes. Changes will be grouped and incorporated at the decision of the CCB. Either a time limit or a quantity of changes may be used to make a change to the language.

Configuration control methods require:

1. Identification and complete description of the items under consideration.

2. Control of these items to prevent any unauthorized changes or variations from occurring.
3. Accounting for the items to establish baselines and track authorized changes so that the presently approved version is known and the delta from the previous level is known. Changes will include corrections and enhancements.

A common language description shall be contained in the standard definition document described in Section 2. Control of the language shall be the responsibility of the CCB. Procedures will be established to record all proposals, changes, and interpretations to the standard document by the CCB.

4.3 Configuration Control for Compilers

All compilers used by the Federal Government, whether owned or not, must be controlled. Those owned by the Federal Government shall be controlled by common language organization. Those not owned will be controlled by the validation procedures which shall be required for all compilers used on federal projects. In both cases, complete descriptions of the product and accurate configuration information shall be maintained. Compilers shall be validated after each modification. The validation procedures are described in Section 5. Compilers used outside the Federal Government are strongly encouraged to use the available validation facilities.

4.4 Configuration Control for Supporting Software

Supporting programs owned by the LSA will be controlled by establishing procedures to state the configuration and

then managing approved changes. All programs will be recorded and cataloged by the LSA for promoting transferability.

4.5 Configuration Control for Application Programs

Application programs will be controlled by the development agencies. The development agencies shall furnish the LSA with a description of all programs developed. The descriptions will be cataloged by the LSA. The description shall be in a standard abstract form.

Each development agency shall be required to search the catalog prior to developing new application programs in order to use existing, proven software rather than developing more.

Section 5

Compilers

5.1 Production Compilers

The initial production compiler will be implemented in either a machine transportable fashion or in the common language itself. It will be targeted for one of the standard militarized computers so as to provide maximum useability. In addition, there may be a machine transportable simulator for the targeted machine to allow further use. The initial production compiler will be implemented in two sections: first a root component which is target independent; and second, the target machine's code generator.

The production versions of the compiler could be developed by anyone. To carry the DoD certification however, means that the version in question has been validated by the validation agency as meeting the standards of the common language. The validation process is explained in Section 5.2.

The following design factors should be considered by any implementor. The use of the common language as the implementing language for the compiler is encouraged. After 1982, it will be required. In the cases where the validation agency has validated a compiler which both executes on and is targeted for the new compiler's host, the use of the common language is mandated.

5.2 Compiler Validation

The purpose in validating the compilers is to determine the degree of conformance to the language as described in the standard definition document. Conformance will be measured to the syntax and semantics of the language. Common language shall be specified in a manner which promotes validation and decreases the chances for misinterpretation by the developer.

The method of validation shall be to compile and execute a standard series of programs written in the common language to test for correct translation. The test set of programs will validate single language statements as well as sequences. The tests shall be comprehensive for all statements as well as for extremities and crucial cases. The tests shall also represent examples of embedded systems applications. A validation report shall be prepared stating the results of the tests and the resources used.

The testing shall be done by the validation agency. The validation agency will be responsible for preparing the set of standard tests, compiling the validation information, and approving the material. The agency will either officially validate the translator or state necessary corrective actions prior to official validation. The implementor shall be required to state that there are no unauthorized extensions to the language translator.

The published test set shall be recognized as being an incomplete test of the compiler and validation may be denied should the compiler fail any added tests.

As part of the validation effort, benchmark tests to describe compilation speed, compilation memory usage, object code speed, and object code memory usage shall be required.

5.3 Delivery Packages

5.3.1 Compiler Delivery

The delivery package for a new government compiler shall be sent to the validation agency by the implementor. Once the compiler is validated, the delivery package will be forwarded to the LSA by the validation agency thus only validated compilers will be available from the LSA.

The package will consist of the following items.

1. The source code and listing for the compiler.
2. The object and/or executable code of the compiler as appropriate.
3. The system programmer's guide which shall include how to install and maintain the compiler. Also included will be the linkage and external interface specifications.
4. The compiler's logic manual which shall describe the internal design of the compiler.
5. The certificate of validation and the validation report.
6. The compiler user's guide including how to execute the compiler and benchmark results against other compilers both for common language and for other languages.
7. Source code along with object and/or executable code for the run-time support routines for

housekeeping, error detection and recovery, multi-tasking, etc. and a standard library package.

8. Maintenance plan and procedure.

5.3.2 Compiler Delivery to User Site

A user requesting a copy of a compiler will receive a package containing the following items from the LSA.

1. Object code and/or executable code of the compiler and its run-time support routines as appropriate.
2. The system programmer's guide.
3. The compiler user's guide.
4. A procedure for reporting problems.

5.4 Compiler/User Interface

5.4.1 Compiler Inputs

Inputs to a compiler fall into three categories: source statements to be compiled; target machine characteristics; and compiler control and option parameters.

5.4.1.1 Source statements to be compiled

A given compiler may support various formats of source images, every translator shall accept the 80-character card image.

5.4.1.2 Target machine characteristics

The minimum target machine characteristics that a compiler will accept are as follows.

1. Machine model
2. Memory size
3. Special hardware options
4. Peripheral equipment
5. Optional instruction sets available (e.g. MATHPAK)
6. Operating system

While the input source images may contain this information, the compiler shall accept this data from a separate source. This will allow central control of the target machine configuration.

5.4.1.3 Option parameters

Option parameters are inputs that control the environment of the compilers. These include the following.

1. Listing controls
2. Debugging controls such as whether or not to output code for subscript checking, assertion checking, etc.
3. Optimization options

While either the input source or the machine specification may contain this information, the compiler shall accept this data from a separate source. This will allow central control of the compiler's environment.

5.4.2 Compiler Outputs

5.4.2.1 Object code

Object code output of the compiler should be formatted in accordance with one of the standard loaders controlled by the LSA. The object code may also contain information being passed from the compiler to the compiler's run-time support routines for various purposes such as symbolic debugging, formatted dumps, error-checking, etc.

5.4.2.2 Source listings

The compiler shall be capable of producing source-only listings at the user's option. One listing should contain source as input to the compiler and before any conditional compilation statements are processed. This listing should also show, in the same format, any input statements retrieved from a source library by the compiler. A second listing, which shall be at the user's option, is after all library retrievals and conditional compilation statements have been processed. In addition, both these listing formats shall permit the user to show statement number (for error displays), the occurrence of errors, and the block level.

5.4.2.3 Source/object listing

The source/object listing will have the same options as the source listing. In addition, the listing will show the instructions generated by each compiler input statement. This data shall be intermixed with the listing of source statements.

5.4.2.4 Error messages and listings

Compilers shall be required to use the standard diagnostic and warning messages included in the standard definition document. Messages shall be grouped into classes. All error messages shall be unambiguous. The same error message cannot be used for two or more sets of related symptoms. The implementor shall attempt to provide the following with each error: for syntactical errors, the offending symbol and input statement column; for semantic errors, the offending symbol or expression (e.g. UNDEFINED SYMBOL - X).

Although the error messages shall be mixed with the source and/or object listings, there shall also be an error summary listing giving a total count for each error and the statement numbers on which that error occurred.

5.4.2.5 Symbol attribute listing

The symbol attribute listing shall be produced at the option of the user. The attribute listing shall contain a list of all symbols defined and their characteristics such as type, scale, range,

precision, dimensions, statement on which the symbol was defined, its block level, and its scope. The listing shall be in the collating sequence defined in the standard definition document.

5.4.2.6 Cross reference listing

The cross reference listing shall be produced at the option of the user. The cross reference listing shall contain a list of all symbols defined, their block level, the statement on which they were defined, and a list of statements where the symbol is set or used. The listing shall attempt to differentiate between set and used. The listing shall be in the collating sequence defined in the standard definition document.

5.4.2.7 Program structure map

The program structure map shall be produced at the option of the user. The map shall show the structure of the program with regards to blocks where data, procedure, function, or path are declared. In addition, the map shall show any fetches of definitions or input source from a library.

5.4.2.8 Compiler resource usage listing

The compiler shall output a listing showing the amount of computer resources used. Examples are amount of computer time used, percentage and size of symbol table used, etc.

5.4.2.9 Other Listings

The compilers may produce additional listings, as aids, which are results of the new language.

5.4.3 Run Time Outputs

These outputs shall be produced by the compiler's run-time support routines. These routines provide house-keeping, storage management, task/path management, error detection and recovery, mathematical functions, debugging, etc.

5.4.3.1 Run time errors

The run-time displays for errors shall include the subprogram, definition module, or path, the procedure, and the statement number on which the error occurred. The implementor shall attempt to display the offending symbol if any. The display shall also include a trace back of all currently executing or pending procedures, functions, paths, etc. A dump of all active variables shall be at the option of the user.

5.4.3.2 Run-time debugging

The outputs of run-time debugging shall contain information similar to the error display.

5.4.3.3 Run-time resource usage

The compiler's run-time support routines shall, at the option of the user, produce a summary

of computer resources used in the execution of the program. An example is the amount of computer time and storage used.

5.5 Translator Functional Objectives

Translators referred to in this, and the following, section are meant to reference the superset of compilers, interpreters, etc.

The following objectives shall be met by each translator developer. They are to be considered as general guidelines for translator development.

1. The translator shall be validated as error free
2. The translator shall generate efficient code but not sacrifice clarity
3. The translator shall conform to the documentation standards prescribed
4. The translator shall be unforgiving by identifying all syntax and semantic errors
5. The translator shall be supported by high level debugging tools for static testing of programs on the host computer
6. The translator shall assist in enforcing management standards by furnishing required diagnostic messages
7. The translator shall be built in a modular fashion which promotes extensions of CCB approved features

8. The translator shall be designed to interface with library programs and executive programs which all together form an application program development plan. These programs may have been compiled by other compilers

5.6 Translator Production Guidelines

The following items apply to the production of translators. The items are included to address different requirements for various parts of the common language environment.

1. Different types of translators will be produced for different environments. For instance, an interpreter will be produced for purposes of programmer education and for environments where fast response is required. A compiler will be produced for development of application programs
2. A compiler generator program may be developed to speed production of compilers for all target computers
3. Each translator shall have optimization features which may be used to optimize memory useage or execution speed
4. Cross compilers shall be developed to translate common language on a computer having support tools into the machine language of a target computer. The cross compiler may be written in Fortran or in the common language

5. Compilers which have special function target machines, such as graphics, should attempt to make use of the compiler step, rather than the execution step, preparing output to the graphics machine. This capability would be similar to compilation of variables which are used for execution many times.

Section 6

Run-Time Supporting Software

6.1 Purpose and Intent

The purpose of this section is to list the requirements for the executives and operating systems which will be required to support scheduling, execution, and synchronization of application programs written in common language. The intent is to require a basic set of execution time tools which will lead to success of the common language.

Language features to be supported are parallel processing, exception handling, and memory management.

6.2 Executives

Executive programs will be required for each target computer. These should include the following capabilities.

1. Scheduling and execution of application tasks.
Parallel processing shall be allowed to permit a specified number of control paths to operate in parallel and rejoin at a single point either interleaved on one CPU or on multiple CPU's.
2. Handling of interrupts.
3. Handling of I/O requests.
4. Interfacing with a test and debug package.

6.3 Test and Debug Package

Test and debug programs will be required to interface with each executive for all target computers. The following capabilities are required. They may be interactive.

1. Execution and Data Trace - Various levels of trace capabilities shall be included. For instance, the programmer will be able to trace the module or task execution flow through the system or trace all branches through a particular module. Task control blocks for the task being executed should be dumped if specified data is accessed.
2. Memory/Storage Alteration - The operator will have the capability of altering memory or another storage device in symbolic form. All alterations will be entered into a journal to be printed at the end of the test.
3. Journal Print - At any point in the test the operator will be able to print a history of events occurring during the test in time order.

Section 7

Other Supporting Software

7.1 Purpose and Intent

Supporting software described in this section consists of all programs required to specify, design, develop, compile, test (off-line), document, and control application programs written in common language.

Included in this section are those techniques and programs which aid in migration to the common language from present languages.

Where practical, these tools will be written in the common language. A consistent user interface will be required for these tools.

7.2 Compile Time Tools

7.2.1 Fault Detection Tools

Programs to test for facility coding standards, flag error prone constructs, check module interface coordination, and furnish data cross reference listings for programs under development will be required.

7.2.2 Flow Charters

Programs to list the logical structure of a program as well as the control flow of modules within a system

will be required. Various levels of charting will be available to each branch or to show module/subroutine calls.

7.2.3 Correctness Provers

Program verifiers to test in a mathematical fashion, to see that the program corresponds to a given specification in a design language will be required.

7.2.4 Symbolic Program Executors

Programs which logically execute programs under development will be required. These programs will provide a capability to examine symbolic results of an execution in order to compare against the design language specification for correctness.

7.3 Object Program Link/Load Tools

A linking/loading program will be required for each target computer in order to fit the compiled pieces of a software system into an executable system for the particular executive in use. Included in the requirements are some that facilitate system development. These programs may also preserve and check for the strong typing, test assertions, and support other protection mechanisms built into the language. Requirements for the link/load programs are as follows.

1. Convenient User Interface - a command language will be developed to facilitate directing the link/load programs in specifying desired capabilities.
2. Stub Generation - to facilitate top down programming, the link/load program will build stubs for modules

that are not yet defined in a system. Stubs should take specified memory space and have loops generated to take the specified CPU time.

3. Module Placement - module placement in memory will be as specified by the user or may be placed in memory by the link/load program. Amount of memory space used will be optimized if requested.
4. Automatic Module Retrieval/Library Hierarchies - the order of search for object modules may be specified by the user to facilitate retrieval.
5. Object Module Formats/Object File Organization - the formats of the items to be linked and loaded will be recognized by the link/load program so that common language items may be loaded along with non common language items.
6. Symbol Definition - the user will be able to specify directions for the translator for any undefined symbols.
7. Boundary Alignment - the user will be able to specify locations for externally relocatable elements either by specific location or symbolically. Space between elements may be specified either absolutely or as a percentage of the element's space.
8. Automatic Common Placement - the user will be able to place at will or the link/load program will place all common data.
9. Closed Tree Structures - specified portions of a tree structure may be loaded without all called elements.

10. Dynamic Overlays - overlays will be built with the intention of dynamic loading.
11. Multiple Element Placement - the multiple loading of an object element will be allowed.
12. Listing Output - the link/load program should furnish listings for the following.
 - a. Memory map usage (areas used by specific elements and areas not used)
 - b. Cross-reference listing of external symbols by module reference
 - c. Hierarchical charts of element calls and includes
 - d. Stubs generated
 - e. Dynamic overlays
 - f. Comprehensive diagnostics for all link/load capabilities

7.4 Requirements Generation Tools

A method of cataloging system requirements in order to test throughout development for requirement compliance will be provided. This capability will be similar to that of the Problem Statement Language/Problem Statement Analysis (PSL/PSA) programs.

7.5 Design Tools

Design tools required will be those which support a methodical design process. Programs will be available to

support a design language by testing syntax correctness, formatting, and proof of correctness. The designed capabilities will be able to be compared to the requirements.

7.6 Construction Tools

Construction tools which assist in the writing of programs in the common language will be required.

7.6.1 Design Language Translation Tools

Methods will be required to translate programs written in a design language to the common language.

7.6.2 External Library Systems

Library methods of storing and retrieving both source, object code, and test cases will be required. Entries shall include type definitions, input - output packages, common pools of shared declarations, application oriented software packages, other separately compiled segments, and machine configuration specifications. The library shall be structured to allow entries to be associated with particular applications, projects, and users. Multiple versions of library entries shall be allowed.

7.6.3 Test and Debug Systems

Test and debug capabilities will be required for diagnosing the performance of programs before they are integrated into a system. The following capabilities will be required for each test system.

1. Test input specification and control language. This capability is intended to prevent each programmer from having to develop test drivers.

2. Memory dump related to the symbolic program along with register indication.
3. Snap capability for data areas.
4. Trace methods to record paths and frequency of statement execution.
5. Breakpoint and dump capabilities.
6. Symbolic patch methods to be used on conjunction with the library systems.
7. Cumulative recording routines to track which parts of programs have/have not been tested to allow better test case generation.

7.7 Integration Tools

Integration tools which will assist in fitting the individual programs into a system will be required.

7.8 Control Tools

7.8.1 Configuration Management Aids

Programs which will assist in controlling the common language and programs written in common language will be required.

7.8.2 Project Control Tools

Programs and techniques to control project resources and track status will be required.

7.9 Migration Tools

Programs which will aid the migration of present languages to the common language by performing translations may be required.

Section 8

Application Software

One of the goals in the use of a Higher Order Language is to increase portability of programs written in the language. For those application programs written in the common language portability will be promoted by the following methods.

1. Information concerning application programs will be maintained by the Language Support Agency and cataloged by type of program. A standard abstract format will be employed.
2. Major types of embedded systems will be identified and basic tasks within these types will be identified for catalog purposes.

The types of embedded systems are as follows:

- a. Command and Control
- b. Communications
- c. Avionics
- d. Shipboard
- e. Test Equipment

3. Organizations concerned with common language will encourage special interest groups within user organizations to address each of the major types of embedded systems as well as common functions across all embedded systems. Groups would be formed for communication, avionics, command and control, operating systems, data management and software development.

Section 9

Language and Environment Documentation

Documentation goals for the common language and its environment are to describe fully all aspects and have the descriptions available in many forms. The Language Support Agency (LSA) shall be responsible for setting documentation standards.

9.1 Language Documentation

Basic documentation for the language, which will take precedence over all other language documents will be the standard definition document. The standard definition document shall include the syntax, semantics, and appropriate examples of each language feature including those for standard library definitions. Other documents required to support the language as follows.

1. Users Guide
2. Primer
3. Translator Developers Guide

These documents shall be independent of any particular implementation of the common language.

9.2 Compiler Documentation

A basic set of documentation will be required for each compiler and will include the following.

1. Users Guide
2. Logic Manual
3. Compiled examples
4. Benchmark results
5. Compiler defined parameters
6. Techniques for effective use of this compiler.

9.3 Supporting/Application Software Documentation

The LSA will develop standards for documentation including the definition of minimum standards for supporting software and application programs.

9.4 Methods of Documentation

A master index will be maintained for all documentation pertaining to the common language. Abstracts will be cataloged to encourage developers to use existing, documented programs. The primary method of documenting all software will be the listings.

Formats for all levels of program documentation shall be defined.

All user interface control language shall be specified in a formal method, such as, Backus-Naur format.

Documentation will be provided in languages other than English. These languages will include French, German, Italian, Portuguese, Japanese, Spanish, Arabic, Turkish, Greek, Norwegian and others.

Section 10

Information Collection, Dissemination, and Promotion

It will be the responsibility of the Language Support Agency (LSA) to collect and disseminate all information concerning the common language.

The LSA will maintain information about the language as well as programs written in the language and those which support the language. This information will be cataloged into a hierarchical document which will contain sections on all types of documents which pertain to the common language. The catalog will contain a brief description of each item of documentation in the form of a standard abstract. Each description will include title, purpose, author, revision level, size, and key words. The catalog will also include a Key-Word-In-Context (KWIC) listing for search purposes.

The LSA will maintain statistical information about the use of the common language. Statistics will include the number of projects using the language, number of translators, and number of computers for both host and target. Reports from the field shall include information about the detail use of the language and translators. The information will include error studies, difficult to use constructs, amount of machine code used and reasons why, and all proposals for super sets and upgrades. These statistics will be published periodically as part of a common language report which will be the responsibility of the HOLWG. This report will include the present status and plans for the language.

All of this information will be made available to the common language community to ensure that all users and potential users are working with accurate, current information.

A periodic language bulletin will be considered, possibly to be placed on the ARPANET.

Section 11

Training Support

Initial training will be required for programmers using the language, developers of translators, and management. Preparation of courses for each of the various levels will be required. Different modes of training will also be required due to diverse locations, schedules, and background of those requiring training.

11.1 Types of Training Required

11.1.1 Programmers Using the Common Language

Training will be provided for all programmers who will write programs in the common language. This training must consider new as well as experienced programmers and will consist of beginning, intermediate and advanced levels. Refresher courses will also be provided.

Training will be required for language use as well as tool use. Language aspects which help accomplish project objectives such as reliability, efficient memory usage, efficient Central Processor Usage, maintainability, and standard styles should be taught.

Training aids will require manuals for programmers familiar with other HOL's. For instance, the following will be required:

1. Common Language for Jovial Users

2. Common Language for CMS-2 Users
3. Common Language for TACPOL Users
4. Common Language for SPL/I Users

11.1.2 Translator Developers

Training will be provided in the syntax and semantics of the language for personnel developing translators.

11.1.3 Management of Projects Using the Common Language

The management of projects using the language will require overview training for the language and its environment. Training in techniques which promote success in project development should also be prepared.

11.2 Training Modes

Methods of training will include the following.

1. Classroom Instruction
2. Video Tape Courses
3. Computer Terminal Teaching Sessions
4. Self-Instruction Manuals

Material for all of these methods will include liberal use of programming examples with various levels of complexity and will depict the required steps in arriving at a solution.

For the classroom sessions, teaching materials will be required for conducting different levels of training. These materials will aid in scheduling, teaching, and examination of students. Charts for consistent training and class projects should be included.

Materials shall be provided for teaching in languages other than English.

Section 12

Project Management Aids

The purpose of this section is to state requirements for project management aids which are necessary for the success of projects using the common language. These aids consist of tools and techniques which assist in the specification, designing, building, validating and controlling of the software. The tools and techniques are listed in previous sections. This section will relate them to the phases of development.

Different aids are required for each type of activity in the development process. These aids and necessary training in their use are described below.

12.1 Requirements Definition and Tracking Aids

The first and most critical phase of a software project is the specification of requirements. Management must have software tools which allow the identification of all requirements and the tracing of requirements through the project in order to ensure correct mappings onto programs. These tools should also furnish information to judge the completeness and consistency of the requirements and should build a data base which flows through the other phases of the project.

A tool, such as the PSL/PSA, and techniques for testing completeness, such as inspection/reviews, should be used.

12.2 Design Aids

The elements of the design phase consist of recognition and understanding of the latest methods of designing reliable software and the software tools to support their use.

These software tools should lead to a complete and consistent mapping of the requirements onto a structure of programs.

The following tools and ideas should be employed in this phase.

1. Top down development with data structure considerations
2. Explicit design guidelines and standards
3. Requirements tracing through the software structure
4. Planning for integration
5. Planning for change to design and code
6. Validation of each design step by reviews, walk-through or simulation

12.3 Construction Aids

Construction aids consist of all the tools and techniques required to build the software components which will implement the requirements. A complete facility such as a Program Development System or Software Factory should be considered to support development.

The following aids should be available to assist in this project phase.

1. Source entry and modification systems
2. Library control systems to store various versions of programs and retain periodic versions as backup for security purposes
3. Interpretive translation for quick development of algorithms
4. Text editing capabilities with automatic indentation and spacing to promote readability
5. Static test tools for tracing control paths and performing symbolic executions

12.4 Integration Aids

Integration aids consist of all the tools and techniques required to fit the software components together into a validated system.

The following aids should be available to assist in the integration phase.

1. Top down development tools to allow entry of program stubs and replacement by actual programs
2. Status reporting systems for tracking status of programs and troubles
3. Test data generation methods for developing the minimum number of cases for maximum coverage
4. Test status tools for identifying programs and portions of programs that are not yet tested

12.5 Control Aids

Control aids consist of the tools and techniques which allow project management to control all project resources: dollars, people, computers, and lines of code.

The following aids should be available to assist project management.

1. Cost accounting systems to track actual versus budget cost in a work breakdown structure
2. Schedule reporting systems
3. Configuration control systems which track configuration levels and status of changes
4. Control over the use of language features (e.g. GOTO's and recursion)

Workshop on Environment & Control of DOD
Common High Order Language

University of California, Irvine
June 20 - 22

LIST OF PARTICIPANTS

Capt. G. E. Anderson, MCTSSA, Camp Pendleton
Robert Anderson, The Rand Corporation
Robert Balzer, USC - Information Sciences Institute
Capt. Jim Bladen, USAF Armament Lab.
Kenneth L. Bowles, UC - San Diego
Ruven Brooks, UC - Irvine
Thomas E. Cheatham, Harvard University
Paul M. Cohen, Defense Communications Agency
Cdr. John D. Cooper, Chief of Navy Material
Steve Crocker, USC - Information Sciences Institute
Joe Cross, Sperry-Univac
Sam DiNitto, USAF-RADC
Mike Dyer, IBM Corporation
Patricia L. Eddy, Sperry-Univac
Peter J. Elzer, Institute for Defense Analyses
John W. Esch, Sperry-Univac
Peggy Eastwood, McDonnell Douglas
Arthur Evans, Jr., Bolt, Beranek & Newman
Steve Fickas, UC - Irvine
Norm Finn, Rolm Corporation
Dr. David A. Fisher, Institute for Defense Analyses

Gene Fisher, Jet Propulsion Laboratory

Peter Freeman, SofTech, Inc.

Anthony Gargaro, Computer Science Corporation

Susan L. Gerhart, USC - Information Sciences Institute

Robert Glass, Boeing Aerospace Corporation

Harmut Huber, Naval Surface Weapons Center

Al Irvine, SofTech, Inc.

Rob Kling, UC - Irvine

John C. Knight, NASA Langley Research Center

Charles L. Lawson, Jet Propulsion Lab

Warren E. Loper, Naval Ocean Systems Center

David Loveman, Massachusetts Computer Associates

David C. Luckham, Stanford University

Neil Ludham, UC - Los Angeles

John Machado, Naval Electronics Systems

Clem McGowan, SofTech, Inc.

Jim Meehan, UC - Irvine

James Michner, Intermetrics

Duncan E. Morrill, Interstate Electronics Corporation

Robert Morris, Bell Laboratories

John Morison, R.S.R.E.

Mat Myszewski, Massachusetts Computer Associates

Eldred C. Nelson, TRW

Cmdr. Ron Ohlander, Naval Electronics Systems

James E. Prescott, IBM Corporation

Patricia Santoni, Naval Ocean Systems Center

Victor Schneider, The Aerospace Corporation

John T. Shen, Naval Ocean Systems Center

Ann Marmor-Squires, Bolt, Beranek & Newman

Steve Squires, Harvard University

Thomas A. Standish, UC - Irvine

Henry G. Stuebing, Naval Air Development Center

Edward Taft, Xerox - PARC

Richard Taylor, Boeing Computer Services

Warren Teitelman, Xerox - PARC

Dennis Turner, US Army

Peter Wegner, Brown University

Lt. Col. William A. Whitaker, DARPA

Martin Wolfe, U.S. Army

Ray Young, Sperry-Univac